



# Programming POE

**Matt Sergeant**

**Slides at <http://axkit.org/docs/presentations/tpc2002/>**



## What Is POE?

**<purl> poe is a framework for multitasking and networking in plain Perl. See: <http://poe.perl.org/> or a musician (<http://p-o-e.com/>) or the guy who draws (not always work-safe) <http://exploitationnow.com/>**



# Multitasking + Networking





# Multitasking + Networking

- Events



# Multitasking + Networking

- Events
  - ◆ Event Generators



# Multitasking + Networking

- Events
  - ◆ Event Generators
  - ◆ Event Handlers



# Multitasking + Networking

- Events
  - ◆ Event Generators
  - ◆ Event Handlers
  - ◆ Event Queue



# Multitasking + Networking

- Events
  - ◆ Event Generators
  - ◆ Event Handlers
  - ◆ Event Queue
  - ◆ Event Loop



# Multitasking + Networking

- Events
  - ◆ Event Generators
  - ◆ Event Handlers
  - ◆ Event Queue
  - ◆ Event Loop
- User Space *"Threads"*



# Multitasking + Networking

- Events
  - ◆ Event Generators
  - ◆ Event Handlers
  - ◆ Event Queue
  - ◆ Event Loop
- User Space *"Threads"*
  - ◆ A framework for building *"multi-threaded"* applications



# Multitasking + Networking

- Events
  - ◆ Event Generators
  - ◆ Event Handlers
  - ◆ Event Queue
  - ◆ Event Loop
- User Space *"Threads"*
  - ◆ A framework for building *"multi-threaded"* applications
  - ◆ Pure Perl implementation - so threads are really *pseudo threads*



# Multitasking + Networking

- Events
  - ◆ Event Generators
  - ◆ Event Handlers
  - ◆ Event Queue
  - ◆ Event Loop
- User Space *"Threads"*
  - ◆ A framework for building *"multi-threaded"* applications
  - ◆ Pure Perl implementation - so threads are really *pseudo threads*
- Networking



# Multitasking + Networking

- Events
  - ◆ Event Generators
  - ◆ Event Handlers
  - ◆ Event Queue
  - ◆ Event Loop
- User Space *"Threads"*
  - ◆ A framework for building *"multi-threaded"* applications
  - ◆ Pure Perl implementation - so threads are really *pseudo threads*
- Networking
  - ◆ POE isn't just for networking apps...



# Multitasking + Networking

- Events
  - ◆ Event Generators
  - ◆ Event Handlers
  - ◆ Event Queue
  - ◆ Event Loop
- User Space *"Threads"*
  - ◆ A framework for building *"multi-threaded"* applications
  - ◆ Pure Perl implementation - so threads are really *pseudo threads*
- Networking
  - ◆ POE isn't just for networking apps...
  - ◆ But it is great for them.



# Multitasking + Networking

- Events
  - ◆ Event Generators
  - ◆ Event Handlers
  - ◆ Event Queue
  - ◆ Event Loop
- User Space *"Threads"*
  - ◆ A framework for building *"multi-threaded"* applications
  - ◆ Pure Perl implementation - so threads are really *pseudo threads*
- Networking
  - ◆ POE isn't just for networking apps...
  - ◆ But it is great for them.
- POE maintains a sea of objects - like an OS

# Multitasking + Networking

- Events
  - ◆ Event Generators
  - ◆ Event Handlers
  - ◆ Event Queue
  - ◆ Event Loop
- User Space *"Threads"*
  - ◆ A framework for building *"multi-threaded"* applications
  - ◆ Pure Perl implementation - so threads are really *pseudo threads*
- Networking
  - ◆ POE isn't just for networking apps...
  - ◆ But it is great for them.
- POE maintains a sea of objects - like an OS
- Events + User space threads + Select I/O + Components

# Captain POE!





# Why Is This Useful?



# Why Is This Useful?

- Perl Threads Sucked Prior to 5.8



# Why Is This Useful?

- Perl Threads Sucked Prior to 5.8
- POE's threading model is mature and stable



# Why Is This Useful?

- Perl Threads Sucked Prior to 5.8
- POE's threading model is mature and stable
- No Locking Issues

# Why Is This Useful?

- Perl Threads Sucked Prior to 5.8
- POE's threading model is mature and stable
- No Locking Issues
  - ◆ Though there are still some threads *gotchas*

# Why Is This Useful?

- Perl Threads Sucked Prior to 5.8
- POE's threading model is mature and stable
- No Locking Issues
  - ◆ Though there are still some threads *gotchas*
- POE allows you to write usable and *portable* multitasking applications



# Why Use POE?





# Why Use POE?

- Programming is all about abstraction



# Why Use POE?

- Programming is all about abstraction
- POE is an abstraction that makes certain classes of programming easier

# Why Use POE?

- Programming is all about abstraction
- POE is an abstraction that makes certain classes of programming easier
  - ◆ Network Applications

# Why Use POE?

- Programming is all about abstraction
- POE is an abstraction that makes certain classes of programming easier
  - ◆ Network Applications
  - ◆ Applications with long-running system calls



# Why Use POE?

- Programming is all about abstraction
- POE is an abstraction that makes certain classes of programming easier
  - ◆ Network Applications
  - ◆ Applications with long-running system calls
  - ◆ Daemons and other long-running applications

# Why Use POE?

- Programming is all about abstraction
- POE is an abstraction that makes certain classes of programming easier
  - ◆ Network Applications
  - ◆ Applications with long-running system calls
  - ◆ Daemons and other long-running applications
  - ◆ Finite State Machines

# Why Use POE?

- Programming is all about abstraction
- POE is an abstraction that makes certain classes of programming easier
  - ◆ Network Applications
  - ◆ Applications with long-running system calls
  - ◆ Daemons and other long-running applications
  - ◆ Finite State Machines
  - ◆ Applications that must deal with signals or external events

# Why Use POE?

- Programming is all about abstraction
- POE is an abstraction that makes certain classes of programming easier
  - ◆ Network Applications
  - ◆ Applications with long-running system calls
  - ◆ Daemons and other long-running applications
  - ◆ Finite State Machines
  - ◆ Applications that must deal with signals or external events
- POE has a large user community, and many pre-built components

# Why Use POE?

- Programming is all about abstraction
- POE is an abstraction that makes certain classes of programming easier
  - ◆ Network Applications
  - ◆ Applications with long-running system calls
  - ◆ Daemons and other long-running applications
  - ◆ Finite State Machines
  - ◆ Applications that must deal with signals or external events
- POE has a large user community, and many pre-built components
- POE is like threads, but safe and easy!

# Why Use POE?

- Programming is all about abstraction
- POE is an abstraction that makes certain classes of programming easier
  - ◆ Network Applications
  - ◆ Applications with long-running system calls
  - ◆ Daemons and other long-running applications
  - ◆ Finite State Machines
  - ◆ Applications that must deal with signals or external events
- POE has a large user community, and many pre-built components
- POE is like threads, but safe and easy!
  - ◆ Well, mostly. More later.

# Why Use POE?

- Programming is all about abstraction
- POE is an abstraction that makes certain classes of programming easier
  - ◆ Network Applications
  - ◆ Applications with long-running system calls
  - ◆ Daemons and other long-running applications
  - ◆ Finite State Machines
  - ◆ Applications that must deal with signals or external events
- POE has a large user community, and many pre-built components
- POE is like threads, but safe and easy!
  - ◆ Well, mostly. More later.
- POE is fun!

# An Example POE Program

```
use POE;
use POE::Wheel::Run;
POE::Session->create(
    inline_states => {
        _start => sub { $_[KERNEL]->yield("download_porn") },
        download_porn => \&download_porn,
        finished_downloading => \&finished,
        err_output => \&err_output,
        output => \&output,
    });

$poe_kernel->run();
exit(0);

sub download_porn {
    my ($kernel, $heap, $ng) = @_[KERNEL, HEAP, ARG0];
    $heap->{nget} = POE::Wheel::Run->new(
        Program => "nget -p $ng -g $ng -r '.*\\. (jpg|bmp|gif|tif|avi|mpg|mpeg|asx|rm)
        CloseEvent => "finished_downloading",
        StdoutEvent => "output",
        StderrEvent => "err_output",
    );
}

sub output { print "$_[ARG0]\n"; }
sub err_output { print STDERR "STDERR: $_[ARG0]\n"; }

sub finished {
    my ($kernel, $heap, $wheelid) = @_[KERNEL, HEAP, ARG0];
    warn "nget finished\n";
    $kernel->delay_set("download_porn", 9000); # start again in 3 hours
}
```

# An Example POE Program

```
use POE;
use POE::Wheel::Run;
POE::Session->create(
    inline_states => {
        _start => sub { $_[KERNEL]->yield("download_porn") }
        download_porn => \&download_porn,
        finished_downloading => \&finished,
        err_output => \&err_output,
        output => \&output,
    });

$poe_kernel->run();
exit(0);

sub download_porn {
    my ($kernel, $heap, $ng) = @_[KERNEL, HEAP, ARG0];
    $heap->{nget} = POE::Wheel::Run->new(
        program => "ngget -p $ng -g $ng -f '\\.(jpg|bmp|gif|tiff|avi|mpg|mpeg|asx|rm)'"
        CloseEvent => "finished_downloading",
        StdoutEvent => "output",
        StderrEvent => "err_output",
    );
}

sub output { print "$_[ARG0]\n"; }
sub err_output { print "STDERR: $_[ARG0]\n"; }

sub finished {
    my ($kernel, $heap, $wheelid) = @_[KERNEL, HEAP, ARG0];
    warn "nget finished\n";
    $kernel->delay_set("download_porn", 9000); # start again in 3 hours
}
```



# Introducing The POE Kernel



# POE inside the Nutshell





# POE inside the Nutshell

- The POE Kernel runs the show

# POE inside the Nutshell

- The POE Kernel runs the show
- The Kernel has an event loop, based on `select()`



# POE inside the Nutshell

- The POE Kernel runs the show
- The Kernel has an event loop, based on `select()`
- The Kernel schedules *events* to *sessions*



# POE inside the Nutshell

- The POE Kernel runs the show
- The Kernel has an event loop, based on `select()`
- The Kernel schedules *events to sessions*
  - ◆ Network events

# POE inside the Nutshell

- The POE Kernel runs the show
- The Kernel has an event loop, based on `select()`
- The Kernel schedules *events to sessions*
  - ◆ Network events
    - Events: connected, closed



# POE inside the Nutshell

- The POE Kernel runs the show
- The Kernel has an event loop, based on `select()`
- The Kernel schedules *events to sessions*
  - ◆ Network events
    - Events: connected, closed
  - ◆ I/O events

# POE inside the Nutshell

- The POE Kernel runs the show
- The Kernel has an event loop, based on `select()`
- The Kernel schedules *events to sessions*
  - ◆ Network events
    - Events: connected, closed
  - ◆ I/O events
    - data available, flushed

# POE inside the Nutshell

- The POE Kernel runs the show
- The Kernel has an event loop, based on `select()`
- The Kernel schedules *events to sessions*
  - ◆ Network events
    - Events: connected, closed
  - ◆ I/O events
    - data available, flushed
  - ◆ Timing events

# POE inside the Nutshell

- The POE Kernel runs the show
- The Kernel has an event loop, based on `select()`
- The Kernel schedules *events to sessions*
  - ◆ Network events
    - Events: connected, closed
  - ◆ I/O events
    - data available, flushed
  - ◆ Timing events
  - ◆ Job control events

# POE inside the Nutshell

- The POE Kernel runs the show
- The Kernel has an event loop, based on `select()`
- The Kernel schedules *events to sessions*
  - ◆ Network events
    - Events: connected, closed
  - ◆ I/O events
    - data available, flushed
  - ◆ Timing events
  - ◆ Job control events
  - ◆ OS Signal events

# POE inside the Nutshell

- The POE Kernel runs the show
- The Kernel has an event loop, based on `select()`
- The Kernel schedules *events to sessions*
  - ◆ Network events
    - Events: connected, closed
  - ◆ I/O events
    - data available, flushed
  - ◆ Timing events
  - ◆ Job control events
  - ◆ OS Signal events
  - ◆ Manually called events

# POE inside the Nutshell

- The POE Kernel runs the show
- The Kernel has an event loop, based on `select()`
- The Kernel schedules *events to sessions*
  - ◆ Network events
    - Events: connected, closed
  - ◆ I/O events
    - data available, flushed
  - ◆ Timing events
  - ◆ Job control events
  - ◆ OS Signal events
  - ◆ Manually called events
- The Kernel also manages resources

# POE inside the Nutshell

- The POE Kernel runs the show
- The Kernel has an event loop, based on `select()`
- The Kernel schedules *events to sessions*
  - ◆ Network events
    - Events: connected, closed
  - ◆ I/O events
    - data available, flushed
  - ◆ Timing events
  - ◆ Job control events
  - ◆ OS Signal events
  - ◆ Manually called events
- The Kernel also manages resources
  - ◆ Uses a reference counting garbage collector



# Kernel Husks

# Kernel Husks

- Kernel event loop can be replaced wholesale with other event loops:

# Kernel Husks

- Kernel event loop can be replaced wholesale with other event loops:
  - ◆ Event.pm

# Kernel Husks

- Kernel event loop can be replaced wholesale with other event loops:
  - ◆ Event.pm
  - ◆ Gtk

# Kernel Husks

- Kernel event loop can be replaced wholesale with other event loops:
  - ◆ Event.pm
  - ◆ Gtk
  - ◆ Tk

# Kernel Husks

- Kernel event loop can be replaced wholesale with other event loops:
  - ◆ Event.pm
  - ◆ Gtk
  - ◆ Tk
  - ◆ poll()

# Kernel Husks

- Kernel event loop can be replaced wholesale with other event loops:
  - ◆ Event.pm
  - ◆ Gtk
  - ◆ Tk
  - ◆ poll()
- The Kernel stops when there is nothing left to do

# Kernel Husks

- Kernel event loop can be replaced wholesale with other event loops:
  - ◆ Event.pm
  - ◆ Gtk
  - ◆ Tk
  - ◆ poll()
- The Kernel stops when there is nothing left to do
- To start a POE application, you run the kernel:

# Kernel Husks

- Kernel event loop can be replaced wholesale with other event loops:
  - ◆ Event.pm
  - ◆ Gtk
  - ◆ Tk
  - ◆ poll()
- The Kernel stops when there is nothing left to do
- To start a POE application, you run the kernel:

```
use POE;  
$poe_kernel->run;  
exit(0);
```



# **Introductory POE - Sessions**



# A Sleeper

# A Sleeper

```
use POE;
$|++;

POE::Session->create(
    inline_states => {
        _start => \&start,
        sleeper => \&sleeper,
    },
    args => [ 1 ],
);

print "Running Kernel\n";
$poe_kernel->run();
print "Exiting\n";
exit(0);

sub start {
    my ($kernel, $time) = @_ [KERNEL, ARG0];
    $kernel->yield("sleeper", $time);
}

sub sleeper {
    my ($kernel, $session, $time)
        = @_ [KERNEL, SESSION, ARG0];
    print "Hello OSCon Attendees!\n";
    $kernel->delay_set("sleeper", $time, $time);
}
```

# A Sleeper

```
use POE;
$|++;

POE::Session->create(
    inline_states => {
        _start => \&start,
        sleeper => \&sleeper,
    },
    args => [ 1 ],
);

print "Running Kernel\n";
$poe_kernel->run();
print "Exiting\n";
exit(0);

sub start {
    my ($kernel, $time) = @_ [KERNEL, ARG0];
    $kernel->yield("sleeper", $time);
}

sub sleeper {
    my ($kernel, $session, $time)
        = @_ [KERNEL, SESSION, ARG0];
    print "Hello OSCon Attendees!\n";
    $kernel->delay_set("sleeper", $time, $time);
}
```

- **Lots** of new concepts here!

# A Sleeper

```
use POE;
$|++;

POE::Session->create(
    inline_states => {
        _start => \&start,
        sleeper => \&sleeper,
    },
    args => [ 1 ],
);

print "Running Kernel\n";
$poe_kernel->run();
print "Exiting\n";
exit(0);

sub start {
    my ($kernel, $time) = @_ [KERNEL, ARG0];
    $kernel->yield("sleeper", $time);
}

sub sleeper {
    my ($kernel, $session, $time)
        = @_ [KERNEL, SESSION, ARG0];
    print "Hello OSCon Attendees!\n";
    $kernel->delay_set("sleeper", $time, $time);
}
```

- ***Lots*** of new concepts here!



# A Sleeper

```
use POE;
$|++;

POE::Session->create(
    inline_states => {
        _start => \&start,
        sleeper => \&sleeper,
    },
    args => [ 1 ],
);

print "Running Kernel\n";
$poe_kernel->run();
print "Exiting\n";
exit(0);

sub start {
    my ($kernel, $time) = @_ [KERNEL, ARG0];
    $kernel->yield("sleeper", $time);
}

sub sleeper {
    my ($kernel, $session, $time)
        = @_ [KERNEL, SESSION, ARG0];
    print "Hello OSCon Attendees!\n";
    $kernel->delay_set("sleeper", $time, $time);
}
```

- Creating a session

# A Sleeper

```
use POE;
$|++;

POE::Session->create(
    inline_states => {
        _start => \&start,
        sleeper => \&sleeper,
    },
    args => [ 1 ],
);

print "Running Kernel\n";
$poe_kernel->run();
print "Exiting\n";
exit(0);

sub start {
    my ($kernel, $time) = @_[KERNEL, ARG0];
    $kernel->yield("sleeper", $time);
}

sub sleeper {
    my ($kernel, $session, $time)
        = @_[KERNEL, SESSION, ARG0];
    print "Hello OSCon Attendees!\n";
    $kernel->delay_set("sleeper", $time, $time);
}
```

- Starting the Kernel

# A Sleeper

```
use POE;
$|++;

POE::Session->create(
    inline_states => {
        _start => \&start,
        sleeper => \&sleeper,
    },
    args => [ 1 ],
);

print "Running Kernel\n";
$poe_kernel->run();
print "Exiting\n";
exit(0);

sub start {
    my ($kernel, $time) = @_ [KERNEL, ARG0];
    $kernel->yield("sleeper", $time);
}

sub sleeper {
    my ($kernel, $session, $time)
        = @_ [KERNEL, SESSION, ARG0];
    print "Hello OSCon Attendees!\n";
    $kernel->delay_set("sleeper", $time, $time);
}
```

- Event Handlers



# New Concepts





# New Concepts

- Creating Sessions



# New Concepts

- Creating Sessions

```
POE::Session->create(...);
```

# New Concepts

- Creating Sessions

```
POE::Session->create(...);
```

- Starting the Kernel



# New Concepts

- Creating Sessions

```
POE::Session->create(...);
```

- Starting the Kernel

```
$poe_kernel->run();
```



# New Concepts

- Creating Sessions

```
POE::Session->create(...);
```

- Starting the Kernel

```
$poe_kernel->run();
```

- Event Handlers

# New Concepts

- Creating Sessions

```
POE::Session->create(...);
```

- Starting the Kernel

```
$poe_kernel->run();
```

- Event Handlers

- Event Handler Parameters

# New Concepts

- Creating Sessions

```
POE::Session->create(...);
```

- Starting the Kernel

```
$poe_kernel->run();
```

- Event Handlers

- Event Handler Parameters

```
my ($kernel, $session, $time) = @_[KERNEL, SESSION, ARG0];
```

# New Concepts

- Creating Sessions

```
POE::Session->create(...);
```

- Starting the Kernel

```
$poe_kernel->run();
```

- Event Handlers

- Event Handler Parameters

```
my ($kernel, $session, $time) = @_[KERNEL, SESSION, ARG0];
```

- ◆ This is an array slice

# New Concepts

- Creating Sessions

```
POE::Session->create(...);
```

- Starting the Kernel

```
$poe_kernel->run();
```

- Event Handlers

- Event Handler Parameters

```
my ($kernel, $session, $time) = @_[KERNEL, SESSION, ARG0];
```

- ◆ This is an array slice

- ◆ KERNEL, SESSION and ARG0 are constants exported by POE.pm

# New Concepts

- Creating Sessions

```
POE::Session->create(...);
```

- Starting the Kernel

```
$poe_kernel->run();
```

- Event Handlers

- Event Handler Parameters

```
my ($kernel, $session, $time) = @_[KERNEL, SESSION, ARG0];
```

- ◆ This is an array slice

- ◆ KERNEL, SESSION and ARG0 are constants exported by POE.pm

- Calling Events

# New Concepts

- Creating Sessions

```
POE::Session->create(...);
```

- Starting the Kernel

```
$poe_kernel->run();
```

- Event Handlers

- Event Handler Parameters

```
my ($kernel, $session, $time) = @_[KERNEL, SESSION, ARG0];
```

- ◆ This is an array slice

- ◆ KERNEL, SESSION and ARG0 are constants exported by POE.pm

- Calling Events

```
$kernel->yield("sleeper", $time);
```

# New Concepts

- Creating Sessions

```
POE::Session->create(...);
```

- Starting the Kernel

```
$poe_kernel->run();
```

- Event Handlers

- Event Handler Parameters

```
my ($kernel, $session, $time) = @_[KERNEL, SESSION, ARG0];
```

- ◆ This is an array slice

- ◆ KERNEL, SESSION and ARG0 are constants exported by POE.pm

- Calling Events

```
$kernel->yield("sleeper", $time);
```

```
$kernel->delay_set("sleeper", $time, $time);
```

# What is A Session?



# What is A Session?

- A Session is an Object floating around in the Kernel



# What is A Session?

- A Session is an Object floating around in the Kernel
- A Session encapsulates:



# What is A Session?

- A Session is an Object floating around in the Kernel
- A Session encapsulates:
  - ◆ Private User Data (the *heap*)



# What is A Session?

- A Session is an Object floating around in the Kernel
- A Session encapsulates:
  - ◆ Private User Data (the *heap*)
  - ◆ Events (A.K.A. "States")



# What is A Session?

- A Session is an Object floating around in the Kernel
- A Session encapsulates:
  - ◆ Private User Data (the *heap*)
  - ◆ Events (A.K.A. "States")
- The Kernel calls a session's events...



# What is A Session?

- A Session is an Object floating around in the Kernel
- A Session encapsulates:
  - ◆ Private User Data (the *heap*)
  - ◆ Events (A.K.A. "States")
- The Kernel calls a session's events...
  - ◆ When network data or a connection is waiting



# What is A Session?

- A Session is an Object floating around in the Kernel
- A Session encapsulates:
  - ◆ Private User Data (the *heap*)
  - ◆ Events (A.K.A. "States")
- The Kernel calls a session's events...
  - ◆ When network data or a connection is waiting
  - ◆ When a timeout/alarm occurs



# What is A Session?

- A Session is an Object floating around in the Kernel
- A Session encapsulates:
  - ◆ Private User Data (the *heap*)
  - ◆ Events (A.K.A. "States")
- The Kernel calls a session's events...
  - ◆ When network data or a connection is waiting
  - ◆ When a timeout/alarm occurs
  - ◆ In response to an OS signal



# What is A Session?

- A Session is an Object floating around in the Kernel
- A Session encapsulates:
  - ◆ Private User Data (the *heap*)
  - ◆ Events (A.K.A. "States")
- The Kernel calls a session's events...
  - ◆ When network data or a connection is waiting
  - ◆ When a timeout/alarm occurs
  - ◆ In response to an OS signal
  - ◆ For a GUI event (when using Gtk or Tk event loops)



# What is A Session?

- A Session is an Object floating around in the Kernel
- A Session encapsulates:
  - ◆ Private User Data (the *heap*)
  - ◆ Events (A.K.A. "States")
- The Kernel calls a session's events...
  - ◆ When network data or a connection is waiting
  - ◆ When a timeout/alarm occurs
  - ◆ In response to an OS signal
  - ◆ For a GUI event (when using Gtk or Tk event loops)
  - ◆ When an event is manually called





# Sessions as *Thread Objects*



# Sessions as *Thread Objects*

- Sessions are almost like thread objects



# Sessions as *Thread Objects*

- Sessions are almost like thread objects
  - ◆ A thread object is an object which has a thread of it's own in multithreaded applications



# Sessions as *Thread Objects*

- Sessions are almost like thread objects
  - ◆ A thread object is an object which has a thread of it's own in multithreaded applications
- To allow another event to run you must return from your function

# Sessions as *Thread Objects*

- Sessions are almost like thread objects
  - ◆ A thread object is an object which has a thread of it's own in multithreaded applications
- To allow another event to run you must return from your function

```
use POE;
$|++;

POE::Session->create(
    inline_states => {
        _start => \&start,
        run_later => \&run_later,
    },
    args => [ 3 ],
);
$poe_kernel->run();
exit(0);

sub start {
    my ($kernel, $time) = @_ [KERNEL, ARG0];
    print "Session started\n";
    $kernel->yield("run_later");
    sleep($time);
}

sub run_later {
    print "Hello OSCon Attendees!\n";
}
```

# Sessions as *Thread Objects*

- Sessions are almost like thread objects
  - ◆ A thread object is an object which has a thread of it's own in multithreaded applications
- To allow another event to run you must return from your function

```
use POE;  
$|++;
```

```
POE::Session->create(  
    inline_states => {  
        _start => \&start,  
        run_later => \&run_later,  
    },  
    args => [ 3 ],  
);  
$poe_kernel->run();  
exit(0);
```

```
sub start {  
    my ($kernel, $time) = @_[KERNEL, ARG0];  
    print "Session started\n";  
    $kernel->yield("run_later");  
    sleep($time);  
}
```

← sleeper won't run here

```
sub run_later {  
    print "Hello OSCon Attendees!\n";  
}
```

# Sessions as *Thread Objects*

- Sessions are almost like thread objects
  - ◆ A thread object is an object which has a thread of it's own in multithreaded applications
- To allow another event to run you must return from your function

```
use POE;  
$|++;
```

```
POE::Session->create(  
    inline_states => {  
        _start => \&start,  
        run_later => \&run_later,  
    },  
    args => [ 3 ],  
);  
$poe_kernel->run();  
exit(0);
```

```
sub start {  
    my ($kernel, $time) = @_[KERNEL, ARG0];  
    print "Session started\n";  
    $kernel->yield("run_later");  
    sleep($time);  
}
```

← sleeper won't run here

← sleeper runs when the sleep() returns

```
sub run_later {  
    print "Hello OSCon Attendees!\n";  
}
```

# Sessions as *Thread Objects*

- Sessions are almost like thread objects
  - ◆ A thread object is an object which has a thread of it's own in multithreaded applications
- To allow another event to run you must return from your function

```
use POE;  
$|++;
```

```
POE::Session->create(  
    inline_states => {  
        _start => \&start,  
        run_later => \&run_later,  
    },  
    args => [ 3 ],  
);  
$poe_kernel->run();  
exit(0);
```

```
sub start {  
    my ($kernel, $time) = @_[KERNEL, ARG0];  
    print "Session started\n";  
    $kernel->yield("run_later");  
    sleep($time);  
}
```

← sleeper won't run here

← sleeper runs when the sleep() returns

```
sub run_later {  
    print "Hello OSCon Attendees!\n";  
}
```





# Session Construction



# Session Construction

- POE::Session->create()

# Session Construction

- POE::Session->create()
- All events explicitly declared



# Session Construction

- POE::Session->create()
- All events explicitly declared
- inline\_states :

# Session Construction

- POE::Session->create()
- All events explicitly declared
- inline\_states :

```
POE::Session->create(  
    inline_states => {  
        state_name => \&coderef,  
        ...  
    });
```

# Session Construction

- POE::Session->create()
- All events explicitly declared
- inline\_states :

```
POE::Session->create(  
    inline_states => {  
        state_name => \&coderef,  
        ...  
    });
```

- Arguments passed to \_start event

# Session Construction

- POE::Session->create()
- All events explicitly declared
- inline\_states :

```
POE::Session->create(
  inline_states => {
    state_name => \&coderef,
  });
```

- Arguments passed to \_start event

```
POE::Session->create(
  inline_states => { _start => \&start }
  args => [ 1, "two", "III" ],
);
sub start {
  my (@args) = @_ [ARG0..$#_];
  print "Args: ", join(", ", @args), "\n";
}
```

# Session Construction

- POE::Session->create()
- All events explicitly declared
- inline\_states :

```
POE::Session->create(  
  inline_states => {  
    state_name => \&coderef,  
  }...  
);
```

- Arguments passed to \_start event

```
POE::Session->create(  
  inline_states => { _start => \&start }  
  args => [ 1, "two", "III" ],  
);  
sub start {  
  my (@args) = @_ [ARG0..$#_];  
  print "Args: ", join(", ", @args), "\n";  
}
```

Args: 1, two, III



# Predefined Events





# Predefined Events

- `_start`



# Predefined Events

- `_start`
  - ◆ Called as soon as the session is created



# Predefined Events

- `_start`
  - ◆ Called as soon as the session is created
  - ◆ Used for naming a session (aliasing), initiating a loop, creating network connections, opening files, etc

# Predefined Events

- `_start`
  - ◆ Called as soon as the session is created
  - ◆ Used for naming a session (aliasing), initiating a loop, creating network connections, opening files, etc
  - ◆ **Receives all args passed to `POE::Session->create(args => [...])` in `@_[ARG0..$#_]`**

# Predefined Events

- `_start`
  - ◆ Called as soon as the session is created
  - ◆ Used for naming a session (aliasing), initiating a loop, creating network connections, opening files, etc
  - ◆ **Receives all args passed to `POE::Session->create(args => [...])` in `@_[ARG0..$#_]`**
- `_stop`

# Predefined Events

- `_start`
  - ◆ Called as soon as the session is created
  - ◆ Used for naming a session (aliasing), initiating a loop, creating network connections, opening files, etc
  - ◆ **Receives all args passed to `POE::Session->create(args => [...])` in `@_[ARG0..$#_]`**
- `_stop`
  - ◆ Called when the session is garbage collected

# Predefined Events

- `_start`
  - ◆ Called as soon as the session is created
  - ◆ Used for naming a session (aliasing), initiating a loop, creating network connections, opening files, etc
  - ◆ **Receives all args passed to `POE::Session->create(args => [...])` in `@_[ARG0..$#_]`**
- `_stop`
  - ◆ Called when the session is garbage collected
  - ◆ Useful for cleanup, e.g. closing files, logging, closing sockets

# Predefined Events

- `_start`
  - ◆ Called as soon as the session is created
  - ◆ Used for naming a session (aliasing), initiating a loop, creating network connections, opening files, etc
  - ◆ **Receives all args passed to `POE::Session->create(args => [...])` in `@_[ARG0..$#_]`**
- `_stop`
  - ◆ Called when the session is garbage collected
  - ◆ Useful for cleanup, e.g. closing files, logging, closing sockets
- Others: `_signal`, `_parent`, `_child`, `_default`



# Event Arguments





# Event Arguments

```
my ($kernel, $session, $arg0, $arg1) = @_[KERNEL, SESSION, ARG0, ARG1];
```

# Event Arguments

```
my ($kernel, $session, $arg0, $arg1) = @_[KERNEL, SESSION, ARG0, ARG1];
```

- Array slice of the @\_ array

# Event Arguments

```
my ($kernel, $session, $arg0, $arg1) = @_[KERNEL, SESSION, ARG0, ARG1];
```

- Array slice of the `@_` array
- Every event receives *lots* of parameters:

# Event Arguments

```
my ($kernel, $session, $arg0, $arg1) = @_[KERNEL, SESSION, ARG0, ARG1];
```

- Array slice of the @\_ array
- Every event receives *lots* of parameters:
  - ◆ A reference to POE's kernel instance.

# Event Arguments

```
my ($kernel, $session, $arg0, $arg1) = @_[KERNEL, SESSION, ARG0, ARG1];
```

- Array slice of the @\_ array
- Every event receives *lots* of parameters:
  - ◆ A reference to POE's kernel instance.
  - ◆ A reference to the session's storage space.

# Event Arguments

```
my ($kernel, $session, $arg0, $arg1) = @_[KERNEL, SESSION, ARG0, ARG1];
```

- Array slice of the @\_ array
- Every event receives *lots* of parameters:
  - ◆ A reference to POE's kernel instance.
  - ◆ A reference to the session's storage space.
  - ◆ A reference to the session itself.

# Event Arguments

```
my ($kernel, $session, $arg0, $arg1) = @_[KERNEL, SESSION, ARG0, ARG1];
```

- Array slice of the @\_ array
- Every event receives *lots* of parameters:
  - ◆ A reference to POE's kernel instance.
  - ◆ A reference to the session's storage space.
  - ◆ A reference to the session itself.
  - ◆ A reference to the event's creator.

# Event Arguments

```
my ($kernel, $session, $arg0, $arg1) = @_[KERNEL, SESSION, ARG0, ARG1];
```

- Array slice of the `@_` array
- Every event receives *lots* of parameters:
  - ◆ A reference to POE's kernel instance.
  - ◆ A reference to the session's storage space.
  - ◆ A reference to the session itself.
  - ◆ A reference to the event's creator.
  - ◆ The start of the event's parameter list.

# Event Arguments

```
my ($kernel, $session, $arg0, $arg1) = @_[KERNEL, SESSION, ARG0, ARG1];
```

- Array slice of the @\_ array
- Every event receives *lots* of parameters:
  - ◆ A reference to POE's kernel instance.
  - ◆ A reference to the session's storage space.
  - ◆ A reference to the session itself.
  - ◆ A reference to the event's creator.
  - ◆ The start of the event's parameter list.
  - ◆ *And more!*

# Event Arguments

```
my ($kernel, $session, $arg0, $arg1) = @_[KERNEL, SESSION, ARG0, ARG1];
```

- Array slice of the @\_ array
- Every event receives *lots* of parameters:
  - ◆ A reference to POE's kernel instance.
  - ◆ A reference to the session's storage space.
  - ◆ A reference to the session itself.
  - ◆ A reference to the event's creator.
  - ◆ The start of the event's parameter list.
  - ◆ *And more!*
- Alternate syntaxes :

# Event Arguments

```
my ($kernel, $session, $arg0, $arg1) = @_[KERNEL, SESSION, ARG0, ARG1];
```

- Array slice of the @\_ array
- Every event receives *lots* of parameters:
  - ◆ A reference to POE's kernel instance.
  - ◆ A reference to the session's storage space.
  - ◆ A reference to the session itself.
  - ◆ A reference to the event's creator.
  - ◆ The start of the event's parameter list.
  - ◆ *And more!*
- Alternate syntaxes :
- Ignoring args

# Event Arguments

```
my ($kernel, $session, $arg0, $arg1) = @_[KERNEL, SESSION, ARG0, ARG1];
```

- Array slice of the @\_ array
- Every event receives *lots* of parameters:
  - ◆ A reference to POE's kernel instance.
  - ◆ A reference to the session's storage space.
  - ◆ A reference to the session itself.
  - ◆ A reference to the event's creator.
  - ◆ The start of the event's parameter list.
  - ◆ *And more!*
- Alternate syntaxes :
- Ignoring args

```
my ($kernel, $heap, undef, undef, @args) = @_;
```

# Event Arguments

```
my ($kernel, $session, $arg0, $arg1) = @_[KERNEL, SESSION, ARG0, ARG1];
```

- Array slice of the `@_` array
- Every event receives *lots* of parameters:
  - ◆ A reference to POE's kernel instance.
  - ◆ A reference to the session's storage space.
  - ◆ A reference to the session itself.
  - ◆ A reference to the event's creator.
  - ◆ The start of the event's parameter list.
  - ◆ *And more!*
- Alternate syntaxes :
  - Ignoring args
  - Direct assignment

```
my ($kernel, $heap, undef, undef, @args) = @_;
```

# Event Arguments

```
my ($kernel, $session, $arg0, $arg1) = @_[KERNEL, SESSION, ARG0, ARG1];
```

- Array slice of the `@_` array
- Every event receives *lots* of parameters:
  - ◆ A reference to POE's kernel instance.
  - ◆ A reference to the session's storage space.
  - ◆ A reference to the session itself.
  - ◆ A reference to the event's creator.
  - ◆ The start of the event's parameter list.
  - ◆ *And more!*
- Alternate syntaxes :
  - Ignoring args
  - Direct assignment

```
my ($kernel, $heap, undef, undef, @args) = @_;
```

```
my $kernel = $_[0];  
my $heap = $_[1];  
my @args = @_[3..$#_];
```

# Event Arguments

```
my ($kernel, $session, $arg0, $arg1) = @_[KERNEL, SESSION, ARG0, ARG1];
```

- Array slice of the @\_ array
- Every event receives *lots* of parameters:
  - ◆ A reference to POE's kernel instance.
  - ◆ A reference to the session's storage space.
  - ◆ A reference to the session itself.
  - ◆ A reference to the event's creator.
  - ◆ The start of the event's parameter list.
  - ◆ *And more!*
- Alternate syntaxes :

- Ignoring args

```
my ($kernel, $heap, undef, undef, @args) = @_;
```

- Direct assignment

```
my $kernel = $_[0];  
my $heap = $_[1];  
my @args = @_[3..$#_];
```

- Thus slices with constants are easier, less error prone, and more readable



# Event Arguments - Values



# Event Arguments - Values

- Predefined Arguments:



# Event Arguments - Values

- Predefined Arguments:
  - ◆ KERNEL

# Event Arguments - Values

- Predefined Arguments:
  - ◆ KERNEL
  - ◆ SESSION



# Event Arguments - Values

- Predefined Arguments:
  - ◆ KERNEL
  - ◆ SESSION
  - ◆ HEAP (hashref where you can store things)



# Event Arguments - Values

- Predefined Arguments:
  - ◆ KERNEL
  - ◆ SESSION
  - ◆ HEAP (hashref where you can store things)
  - ◆ ... and some others covered later



# Event Arguments - Values

- Predefined Arguments:
  - ◆ KERNEL
  - ◆ SESSION
  - ◆ HEAP (hashref where you can store things)
  - ◆ ... and some others covered later
- Arguments passed to the event:

# Event Arguments - Values

- Predefined Arguments:
  - ◆ KERNEL
  - ◆ SESSION
  - ◆ HEAP (hashref where you can store things)
  - ◆ ... and some others covered later
- Arguments passed to the event:
  - ◆ ARG0 .. ARG9 or ARG0 .. \$#\_

# Event Arguments - Values

- Predefined Arguments:
  - ◆ KERNEL
  - ◆ SESSION
  - ◆ HEAP (hashref where you can store things)
  - ◆ ... and some others covered later
- Arguments passed to the event:
  - ◆ ARG0 .. ARG9 or ARG0 .. \$#\_
  - ◆ example:

# Event Arguments - Values

- Predefined Arguments:
  - ◆ KERNEL
  - ◆ SESSION
  - ◆ HEAP (hashref where you can store things)
  - ◆ ... and some others covered later
- Arguments passed to the event:
  - ◆ ARG0 .. ARG9 or ARG0 .. \$#\_
  - ◆ example:

```
$kernel->yield('foo', 42, "Perl", "POE", %hash);  
...  
  
sub foo_handler {  
    my ($kernel, $heap, $number, $string1, $string2, %myhash)  
        = @_[KERNEL, HEAP, ARG0, ARG1, ARG2, ARG3 .. $#_];  
    ...  
}
```



# Calling a Session's Event Handler



# Calling a Session's Event Handler

- ASAP calls
  - ◆ `$kernel->yield("event", @args)`
  - ◆ `$kernel->post($session, "event", @args)`
  - ◆ `$kernel->call($session, "event", @args)` (immediate - skip event loop)

# Calling a Session's Event Handler

- ASAP calls
  - ◆ `$kernel->yield("event", @args)`
  - ◆ `$kernel->post($session, "event", @args)`
  - ◆ `$kernel->call($session, "event", @args)` (immediate - skip event loop)
- Delayed calls
  - ◆ `$kernel->delay_set("event", $seconds, @args)`
  - ◆ `$kernel->alarm_set("event", $epoch_time, @args)`



# Calling a Session's Event Handler - yield





# Calling a Session's Event Handler - yield

- `$kernel->yield()`



# Calling a Session's Event Handler - yield

- `$kernel->yield()`
  - ◆ Calls back to the current session (the kernel knows which session is current)



# Calling a Session's Event Handler - yield

- `$kernel->yield()`
  - ◆ Calls back to the current session (the kernel knows which session is current)
  - ◆ This is the most commonly used event method

# Calling a Session's Event Handler - yield

- `$kernel->yield()`
  - ◆ Calls back to the current session (the kernel knows which session is current)
  - ◆ This is the most commonly used event method

```
use POE;
$|++;

POE::Session->create(
    inline_states => {
        event_1 => \&first_event,
        event_2 => \&second_event,
        _start => sub { $_[KERNEL]->yield("event_1") },
    }
);

$poe_kernel->run();
exit(0);

sub first_event {
    my ($kernel) = @_[KERNEL];
    print "First Event\n";
    $kernel->yield("event_2");
}

sub second_event {
    my ($kernel) = @_[KERNEL];
    print "Second Event\n";
    $kernel->yield("event_1");
}
```

# Calling a Session's Event Handler - yield

- `$kernel->yield()`
  - ◆ Calls back to the current session (the kernel knows which session is current)
  - ◆ This is the most commonly used event method

```
use POE;  
$|++;  
  
POE::Session->create(  
    inline_states => {  
        event_1 => \&first_event,  
        event_2 => \&second_event,  
        _start => sub { $_[KERNEL]->yield("event_1") },  
    }  
);  
  
$poe_kernel->run();  
exit(0);  
  
sub first_event {  
    my ($kernel) = @_[KERNEL];  
    print "First Event\n";  
    $kernel->yield("event_2");  
}  
  
sub second_event {  
    my ($kernel) = @_[KERNEL];  
    print "Second Event\n";  
    $kernel->yield("event_1");  
}
```





# Calling a Session's Event Handler - post





# Calling a Session's Event Handler - post

- `$kernel->post($session, "event", @args)`



# Calling a Session's Event Handler - post

- `$kernel->post($session, "event", @args)`
  - ◆ Calls to any session

# Calling a Session's Event Handler - post

- `$kernel->post($session, "event", @args)`
  - ◆ Calls to any session
  - ◆ `$session` can be an ID, a name (alias) or a reference

# Calling a Session's Event Handler - post

- `$kernel->post($session, "event", @args)`
  - ◆ Calls to any session
  - ◆ `$session` can be an ID, a name (alias) or a reference

```
use POE;
$|++;

for (1..2) {
    POE::Session->create(
        inline_states => {
            event => \&my_event,
            _start => sub { $_[KERNEL]->alias_set("session_$_") },
        });
}

$poe_kernel->post('session_1', "event", "session_2");
$poe_kernel->run();
exit(0);

sub my_event {
    my ($kernel, $session, $next) = @_[KERNEL, SESSION, ARG0];
    print "Event in ", $kernel->alias_list, "\n";
    $kernel->post($next, "event", $session->ID);
}
```

# Calling a Session's Event Handler - post

- `$kernel->post($session, "event", @args)`
  - ◆ Calls to any session
  - ◆ `$session` can be an ID, a name (alias) or a reference

```
use POE;
$|++;

for (1..2) {
    POE::Session->create(
        inline_states => {
            event => \&my_event,
            _start => sub { $_[KERNEL]->alias_set("session_$_") },
        });
}

$poe_kernel->post('session_1', "event", "session_2");
$poe_kernel->run();
exit(0);

sub my_event {
    my ($kernel, $session, $next) = @_[KERNEL, SESSION, ARG0];
    print "Event in ", $kernel->alias_list, "\n";
    $kernel->post($next, "event", $session->ID);
}
```





# Calling a Session's Event Handler - call



# Calling a Session's Event Handler - call

- `$kernel->call($session, "event", @args)`

# Calling a Session's Event Handler - call

- `$kernel->call($session, "event", @args)`
  - ◆ Used when you need to get the return value of an event

# Calling a Session's Event Handler - call

- `$kernel->call($session, "event", @args)`
  - ◆ Used when you need to get the return value of an event
  - ◆ Often using this means you've designed your application wrong ;-)

# Calling a Session's Event Handler - call

- `$kernel->call($session, "event", @args)`
  - ◆ Used when you need to get the return value of an event
  - ◆ Often using this means you've designed your application wrong ;-)
  - ◆ `$kernel->call()` means POE's event loop doesn't get a look in

# Calling a Session's Event Handler - call

- `$kernel->call($session, "event", @args)`
  - ◆ Used when you need to get the return value of an event
  - ◆ Often using this means you've designed your application wrong ;-)
  - ◆ `$kernel->call()` means POE's event loop doesn't get a look in

```
use POE;  
$|++;
```

```
POE::Session->create(  
  inline_states => {  
    _start => sub { $_[KERNEL]->yield('main_event') },  
    main_event => \&main,  
    call_event => \&get_kernsessid,  
  });
```

```
$poe_kernel->run();  
exit(0);
```

```
sub main {  
  my ($kernel, $session) = @_[KERNEL, SESSION];  
  my $kernsessid = $kernel->call($session, 'call_event');  
  print "This kernel+session id is: $kernsessid\n";  
}
```

```
sub get_kernsessid {  
  my ($kernel, $session) = @_[KERNEL, SESSION];  
  return $kernel->ID . ":" . $session->ID;  
}
```

# Calling a Session's Event Handler - call

- `$kernel->call($session, "event", @args)`
  - ◆ Used when you need to get the return value of an event
  - ◆ Often using this means you've designed your application wrong ;-)
  - ◆ `$kernel->call()` means POE's event loop doesn't get a look in

```
use POE;  
$|++;
```

```
POE::Session->create(  
  inline_states => {  
    _start => sub { $_[KERNEL]->yield('main_event') },  
    main_event => \&main,  
    call_event => \&get_kernsessid,  
  });
```

```
$poe_kernel->run();  
exit(0);
```

```
sub main {  
  my ($kernel, $session) = @_[KERNEL, SESSION];  
  my $kernsessid = $kernel->call($session, 'call_event');  
  print "This kernel+session id is: $kernsessid\n";  
}
```

```
sub get_kernsessid {  
  my ($kernel, $session) = @_[KERNEL, SESSION];  
  return $kernel->ID . ":" . $session->ID;  
}
```





# Calling a Session's Event Handler - delays





# Calling a Session's Event Handler - delays

- `$kernel->delay_set("event", $seconds_hence, @args)`



# Calling a Session's Event Handler - delays

- `$kernel->delay_set("event", $seconds_hence, @args)`
  - ◆ Use to call an event some seconds in the future

# Calling a Session's Event Handler - delays

- `$kernel->delay_set("event", $seconds_hence, @args)`
  - ◆ Use to call an event some seconds in the future
  - ◆ Calls events on the current session only

# Calling a Session's Event Handler - delays

- `$kernel->delay_set("event", $seconds_hence, @args)`
  - ◆ Use to call an event some seconds in the future
  - ◆ Calls events on the current session only

```
use POE;  
$|++;
```

```
POE::Session->create(  
  inline_states => {  
    _start => sub { $_[KERNEL]->yield('delayed_event') },  
    delayed_event => \&delayed_event,  
  });
```

```
$poe_kernel->run();  
exit(0);
```

```
sub delayed_event {  
  my ($kernel) = @_[KERNEL];  
  print("Time is: ", scalar(localtime), "\n");  
  $kernel->delay_set("delayed_event", 5);  
}
```

# Calling a Session's Event Handler - delays

- `$kernel->delay_set("event", $seconds_hence, @args)`
  - ◆ Use to call an event some seconds in the future
  - ◆ Calls events on the current session only

```
use POE;  
$|++;
```

```
POE::Session->create(  
  inline_states => {  
    _start => sub { $_[KERNEL]->yield('delayed_event') },  
    delayed_event => \&delayed_event,  
  });
```

```
$poe_kernel->run();  
exit(0);
```

```
sub delayed_event {  
  my ($kernel) = @_[KERNEL];  
  print("Time is: ", scalar(localtime), "\n");  
  $kernel->delay_set("delayed_event", 5);  
}
```





# Delays redux





# Delays redux

- Why not use `sleep()`?

# Delays redux

- Why not use `sleep()`?
  - ◆ Because with `delay_set()` POE's event loop can activate:

# Delays redux

- Why not use sleep()?
  - ◆ Because with delay\_set() POE's event loop can activate:

```
use POE;
$|++;

POE::Session->create(
    inline_states => {
        _start => \&delayed_event,
        delayed_event => \&delayed_event,
    },
    args => ["Session 1", 4],
);
POE::Session->create(
    inline_states => {
        _start => \&delayed_event,
        delayed_event => \&delayed_event,
    },
    args => ["Session 2", 1],
);

$poe_kernel->run();
exit(0);

sub delayed_event {
    my ($kernel, $name, $delay) = @_ [KERNEL, ARG0, ARG1];
    print "$name at ", scalar(localtime), "\n";
    $kernel->delay_set("delayed_event", $delay, $name, $delay);
}
```

# Delays redux

- Why not use sleep()?
  - ◆ Because with delay\_set() POE's event loop can activate:

```
use POE;
$|++;

POE::Session->create(
    inline_states => {
        _start => \&delayed_event,
        delayed_event => \&delayed_event,
    },
    args => ["Session 1", 4],
);
POE::Session->create(
    inline_states => {
        _start => \&delayed_event,
        delayed_event => \&delayed_event,
    },
    args => ["Session 2", 1],
);

$poe_kernel->run();
exit(0);

sub delayed_event {
    my ($kernel, $name, $delay) = @_ [KERNEL, ARG0, ARG1];
    print "$name at ", scalar(localtime), "\n";
    $kernel->delay_set("delayed_event", $delay, $name, $delay);
}
```



# Delays redux

- Why not use sleep()?
  - ◆ Because with delay\_set() POE's event loop can activate:

```
use POE;
$|++;

POE::Session->create(
    inline_states => {
        _start => \&delayed_event,
        delayed_event => \&delayed_event,
    },
    args => ["Session 1", 4],
);
POE::Session->create(
    inline_states => {
        _start => \&delayed_event,
        delayed_event => \&delayed_event,
    },
    args => ["Session 2", 1],
);

$poe_kernel->run();
exit(0);

sub delayed_event {
    my ($kernel, $name, $delay) = @_ [KERNEL, ARG0, ARG1];
    print "$name at ", scalar(localtime), "\n";
    $kernel->delay_set("delayed_event", $delay, $name, $delay);
}
```

- Other things can happen during the delay - like I/O





# Calling a Session's Event Handler - alarms





# Calling a Session's Event Handler - alarms

- `$kernel->alarm_set("timed_event", $time, @args)`

# Calling a Session's Event Handler - alarms

- `$kernel->alarm_set("timed_event", $time, @args)`
  - ◆ Useful for implementing cron-like functionality

# Calling a Session's Event Handler - alarms

- `$kernel->alarm_set("timed_event", $time, @args)`
  - ◆ Useful for implementing cron-like functionality

```

use POE;
use Time::Piece; use Time::Seconds;
$|++;

print "When do you want to rotate the logs? [HH:MM] ";
chomp(my $time_str = <STDIN>);

my $time = localtime->strptime(localtime->ymd.' '.$time_str, "%Y-%m-%d %H:%M");
$time += ONE_DAY if localtime > $time;
print "Will rotate the logs at: ", $time->strftime, "\n";

POE::Session->create(
    inline_states => {
        _start => sub { $_[KERNEL]->alarm_set('rotate_logs', $time->epoch) },
        rotate_logs => \&rotate_logs,
    });
# create more sessions here to do logging stuff
$poe_kernel->run();
exit(0);

sub rotate_logs {
    my ($kernel, $heap) = @_ [KERNEL, HEAP];
    # do the log rotation...
    print "Logs have been rotated\n";
    # re-post the event next time.
    $time += ONE_DAY;
    $kernel->alarm_set('rotate_logs', $time->epoch);
}

```

# Calling a Session's Event Handler - alarms

- `$kernel->alarm_set("timed_event", $time, @args)`
  - ◆ Useful for implementing cron-like functionality

```

use POE;
use Time::Piece; use Time::Seconds;
$|++;

print "When do you want to rotate the logs? [HH:MM] ";
chomp(my $time_str = <STDIN>);

my $time = localtime->strptime(localtime->ymd.' '.$time_str, "%Y-%m-%d %H:%M");
$time += ONE_DAY if localtime > $time;
print "Will rotate the logs at: ", $time->strftime, "\n";

POE::Session->create(
    inline_states => {
        _start => sub { $_[KERNEL]->alarm_set('rotate_logs', $time->epoch) },
        rotate_logs => \&rotate_logs,
    });
# create more sessions here to do logging stuff
$poe_kernel->run();
exit(0);

sub rotate_logs {
    my ($kernel, $heap) = @_[KERNEL, HEAP];
    # do the log rotation...
    print "Logs have been rotated\n";
    # re-post the event next time.
    $time += ONE_DAY;
    $kernel->alarm_set('rotate_logs', $time->epoch);
}

```





# Using the HEAP



# Using the HEAP

- `$_[HEAP]` contains per-session storage

# Using the HEAP

- `$_[HEAP]` contains per-session storage
- By default it is a hashref



# Using the HEAP

- `$_[HEAP]` contains per-session storage
- By default it is a hashref
- Can override it with `POE::Session->create( heap => $thing )`



# Using the HEAP

- `$_[HEAP]` contains per-session storage
- By default it is a hashref
- Can override it with `POE::Session->create( heap => $thing )`
- Use the heap instead of global variables

# Using the HEAP

- `$_[HEAP]` contains per-session storage
- By default it is a hashref
- Can override it with `POE::Session->create( heap => $thing )`
- Use the heap instead of global variables

```
sub counter_sub {  
    my ($kernel, $heap) = @_[KERNEL, HEAP];  
    $heap->{counter}++;  
    print "Counter is $heap->{counter}\n";  
    $kernel->yield('counter');  
}
```



# Session aliasing





# Session aliasing

- Sessions can have a name (or names)



# Session aliasing

- Sessions can have a name (or names)

```
$kernel->alias_set("my_session_name")
```

# Session aliasing

- Sessions can have a name (or names)

```
$kernel->alias_set("my_session_name")
```

- Use the alias in `$kernel->post()` calls

# Session aliasing

- Sessions can have a name (or names)

```
$kernel->alias_set("my_session_name")
```

- Use the alias in `$kernel->post()` calls
  - ◆ Or anywhere you need to refer to the session

# Session aliasing

- Sessions can have a name (or names)

```
$kernel->alias_set("my_session_name")
```

- Use the alias in `$kernel->post()` calls
  - ◆ Or anywhere you need to refer to the session
- Multiple aliases are OK too

# Session aliasing

- Sessions can have a name (or names)

```
$kernel->alias_set("my_session_name")
```

- Use the alias in `$kernel->post()` calls
  - ◆ Or anywhere you need to refer to the session
- Multiple aliases are OK too

- Delete an alias with:

```
$kernel->alias_remove("my_session_name")
```



## Introductory POE - I/O

**<purl> wheels are sort of like viruses... they latch onto sessions and inject states into them. It's sort of a linking technique.**

# Wheels



# Wheels

- *"Oh no, not more confusing terminology!"*



# Wheels

- *"Oh no, not more confusing terminology!"*
- Wheels are POE's I/O abstraction layer



# Wheels

- *"Oh no, not more confusing terminology!"*
- Wheels are POE's I/O abstraction layer
- They are created by sessions, and feed events back to the sessions based on I/O events



# Wheels

- *"Oh no, not more confusing terminology!"*
- Wheels are POE's I/O abstraction layer
- They are created by sessions, and feed events back to the sessions based on I/O events
- e.g. `POE::Wheel::ReadWrite` feeds events when it "can send" data and when it has "data incoming"



# Wheels

- *"Oh no, not more confusing terminology!"*
- Wheels are POE's I/O abstraction layer
- They are created by sessions, and feed events back to the sessions based on I/O events
- e.g. POE::Wheel::ReadWrite feeds events when it "can send" data and when it has "data incoming"
- Wheels don't stand alone - they must be created by and a session and stored in it's HEAP





# Wheels Example - tail -f

# Wheels Example - tail -f

```
use POE;
use POE::Wheel::FollowTail;

$|++;

my $file = @ARGV[0] || die "No file to watch";

POE::Session->create(
    inline_states => {
        _start => \&main,
        got_record => \&got_record,
    },
    args => [ $file ],
);

$poe_kernel->run();
exit(0);

sub main {
    my ($heap, $log_file) = @_[HEAP, ARG0];

    my $watcher = POE::Wheel::FollowTail->new(
        Filename => $log_file,
        InputEvent => "got_record",
    );

    $heap->{watcher} = $watcher;
}

sub got_record {
    my $log_record = $_[ARG0];
    print $log_record, "\n";
}
```

# Wheels Example - tail -f

```
use POE;
use POE::Wheel::FollowTail;

$|++;

my $file = @ARGV[0] || die "No file to watch";

POE::Session->create(
    inline_states => {
        _start => \&main,
        got_record => \&got_record,
    },
    args => [ $file ],
);

$poe_kernel->run();
exit(0);

sub main {
    my ($heap, $log_file) = @_[HEAP, ARG0];

    my $watcher = POE::Wheel::FollowTail->new(
        Filename => $log_file,
        InputEvent => "got_record",
    );

    $heap->{watcher} = $watcher;
}

sub got_record {
    my $log_record = $_[ARG0];
    print $log_record, "\n";
}
```





# Wheels - How to use them



# Wheels - How to use them

- Create a wheel:

# Wheels - How to use them

- Create a wheel:

```
my $watcher = POE::Wheel::FollowTail->new(  
    Filename => $log_file,  
    InputEvent => "got_record",  
);
```

# Wheels - How to use them

- Create a wheel:

```
my $watcher = POE::Wheel::FollowTail->new(  
    Filename => $log_file,  
    InputEvent => "got_record",  
);
```

- Store it in the session's heap or it gets garbage collected:

# Wheels - How to use them

- Create a wheel:

```
my $watcher = POE::Wheel::FollowTail->new(  
    Filename => $log_file,  
    InputEvent => "got_record",  
);
```

- Store it in the session's heap or it gets garbage collected:

```
$heap->{watcher} = $watcher;
```

# Wheels - How to use them

- Create a wheel:

```
my $watcher = POE::Wheel::FollowTail->new(  
    Filename => $log_file,  
    InputEvent => "got_record",  
);
```

- Store it in the session's heap or it gets garbage collected:

```
$heap->{watcher} = $watcher;
```

- Implement your callbacks for the Wheel's I/O events

# Wheels - How to use them

- Create a wheel:

```
my $watcher = POE::Wheel::FollowTail->new(  
    Filename => $log_file,  
    InputEvent => "got_record",  
);
```

- Store it in the session's heap or it gets garbage collected:

```
$heap->{watcher} = $watcher;
```

- Implement your callbacks for the Wheel's I/O events
  - ◆ Be sure to read the particular wheel's documentation

# Wheels - How to use them

- Create a wheel:

```
my $watcher = POE::Wheel::FollowTail->new(  
    Filename => $log_file,  
    InputEvent => "got_record",  
);
```

- Store it in the session's heap or it gets garbage collected:

```
$heap->{watcher} = $watcher;
```

- Implement your callbacks for the Wheel's I/O events
  - ◆ Be sure to read the particular wheel's documentation

```
sub got_record {  
    my $log_record = $_[ARG0];  
    print $log_record, "\n";  
}
```



# Socket Example

# Socket Example

```
use POE;
use POE::Wheel::ReadWrite;
use POE::Wheel::SocketFactory;
use POE::Driver::SysRW;
use POE::Filter::Line;
$|++;

POE::Session->create(
    inline_states => {
        _start => \&start_parent,
        _stop  => \&stop_parent,
        accept_client => \&accept_client,
        accept_failed => \&socket_failed,
    });

$poe_kernel->run();
exit 0;

sub start_parent {
    my ($heap) = @_ [HEAP];

    $heap->{listener} = POE::Wheel::SocketFactory->new(
        BindAddress => '127.0.0.1',
        BindPort    => 5555,
        Reuse       => 'yes',
        SuccessEvent=> 'accept_client',
        FailureEvent=> 'accept_failed',
    );

    print "SERVER: started listening on port 5555\n";
}

sub stop_parent {
    delete $_[HEAP]->{listener};
    print "SERVER: shutting down\n";
}
```

# Socket Example

```
use POE;
use POE::Wheel::ReadWrite;
use POE::Wheel::SocketFactory;
use POE::Driver::SysRW;
use POE::Filter::Line;
$|++;

POE::Session->create(
    inline_states => {
        _start => \&start_parent,
        _stop  => \&stop_parent,
        accept_client => \&accept_client,
        accept_failed => \&socket_failed,
    });

$poe_kernel->run();
exit 0;

sub start_parent {
    my ($heap) = @_ [HEAP];

    $heap->{listener} = POE::Wheel::SocketFactory->new(
        BindAddress => '127.0.0.1',
        BindPort    => 5555,
        Reuse       => 'yes',
        SuccessEvent=> 'accept_client',
        FailureEvent=> 'accept_failed',
    );

    print "SERVER: started listening on port 5555\n";
}

sub stop_parent {
    delete $_[HEAP]->{listener};
    print "SERVER: shutting down\n";
}
```

Create the Wheel

# Socket Example

```
use POE;
use POE::Wheel::ReadWrite;
use POE::Wheel::SocketFactory;
use POE::Driver::SysRW;
use POE::Filter::Line;
$|++;

POE::Session->create(
    inline_states => {
        _start => \&start_parent,
        _stop  => \&stop_parent,
        accept_client => \&accept_client,
        accept_failed => \&socket_failed,
    });

$poe_kernel->run();
exit 0;

sub start_parent {
    my ($heap) = @_[HEAP];

    $heap->{listener} = POE::Wheel::SocketFactory->new(
        BindAddress => '127.0.0.1',
        BindPort    => 5555,
        Reuse       => 'yes',
        SuccessEvent=> 'accept_client',
        FailureEvent=> 'accept_failed',
    );

    print "SERVER: started listening on port 5555\n";
}

sub stop_parent {
    delete $_[HEAP]->{listener};
    print "SERVER: shutting down\n";
}
```



Create the Wheel

Map events to our states



# Socket Example (cont.)



# Socket Example (cont.)

```
sub accept_client {
    my ($kernel, $heap, $socket, $peeraddr, $peerport) = @_[KERNEL, HEAP, ARG0..ARG2];
    print "SERVER: recieved a connection from $peeraddr:$peerport\n";

    POE::Session->create(
        inline_states => {
            _start      => \&start_child,
            socket_input => \&socket_input,
            socket_failed => \&socket_failed,
            check_shutdown => \&check_shutdown,
        },
        args => [ $socket, $peeraddr, $peerport ],
    );
}

sub start_child {
    my ($heap, $socket, $peeraddr, $peerport) = @_[HEAP, ARG0..ARG2];

    $heap->{readwrite} = POE::Wheel::ReadWrite->new(
        Handle => $socket,
        Driver => POE::Driver::SysRW->new(),
        Filter => POE::Filter::Line->new(),
        InputEvent => 'socket_input',
        ErrorEvent => 'socket_failed',
        FlushedEvent => 'check_shutdown',
    );
}

sub check_shutdown {
    delete $_[HEAP]->{readwrite} if ($_[HEAP]->{shutdown_now});
}

sub socket_input {
    my ($heap, $buf) = @_[HEAP, ARG0];
    $heap->{readwrite}->put($buf);
    $heap->{shutdown_now} = 1 if ($buf eq "QUIT");
}

sub socket_failed {
    delete $_[HEAP]->{readwrite};
}
```

# Socket Example (cont.)

```
sub accept_client {
  my ($kernel, $heap, $socket, $peeraddr, $peerport) = @_[KERNEL, HEAP, ARG0..ARG2];
  print "SERVER: recieved a connection from $peeraddr:$peerport\n";
```

```
  POE::Session->create(
    inline_states => {
      _start      => \&start_child,
      socket_input => \&socket_input,
      socket_failed => \&socket_failed,
      check_shutdown => \&check_shutdown,
    },
    args => [ $socket, $peeraddr, $peerport ],
  );
}
```

Create a session for each connection

```
sub start_child {
  my ($heap, $socket, $peeraddr, $peerport) = @_[HEAP, ARG0..ARG2];
```

```
  $heap->{readwrite} = POE::Wheel::ReadWrite->new(
    Handle => $socket,
    Driver => POE::Driver::SysRW->new(),
    Filter => POE::Filter::Line->new(),
    InputEvent => 'socket_input',
    ErrorEvent => 'socket_failed',
    FlushedEvent => 'check_shutdown',
  );
}
```

```
sub check_shutdown {
  delete $_[HEAP]->{readwrite} if ($_[HEAP]->{shutdown_now});
}
```

```
sub socket_input {
  my ($heap, $buf) = @_[HEAP, ARG0];
  $heap->{readwrite}->put($buf);
  $heap->{shutdown_now} = 1 if ($buf eq "QUIT");
}
```

```
sub socket_failed {
  delete $_[HEAP]->{readwrite};
}
```

# Socket Example (cont.)

```
sub accept_client {
  my ($kernel, $heap, $socket, $peeraddr, $peerport) = @_[KERNEL, HEAP, ARG0..ARG2];
  print "SERVER: recieved a connection from $peeraddr:$peerport\n";
```

```
    POE::Session->create(
      inline_states => {
        _start      => \&start_child,
        socket_input => \&socket_input,
        socket_failed => \&socket_failed,
        check_shutdown => \&check_shutdown,
      },
      args => [ $socket, $peeraddr, $peerport ],
    );
}
```

Create a session for each connection

```
sub start_child {
  my ($heap, $socket, $peeraddr, $peerport) = @_[HEAP, ARG0..ARG2];
```

```
  $heap->{readwrite} = POE::Wheel::ReadWrite->new(
    Handle => $socket,
    Driver => POE::Driver::SysRW->new(),
    Filter => POE::Filter::Line->new(),
    InputEvent => 'socket_input',
    ErrorEvent => 'socket_failed',
    FlushedEvent => 'check_shutdown',
  );
}
```

Use a ReadWrite Wheel for the connection

```
sub check_shutdown {
  delete $_[HEAP]->{readwrite} if ($_[HEAP]->{shutdown_now});
}
```

```
sub socket_input {
  my ($heap, $buf) = @_[HEAP, ARG0];
  $heap->{readwrite}->put($buf);
  $heap->{shutdown_now} = 1 if ($buf eq "QUIT");
}
```

```
sub socket_failed {
  delete $_[HEAP]->{readwrite};
}
```

# Socket Example (cont.)

```
sub accept_client {
  my ($kernel, $heap, $socket, $peeraddr, $peerport) = @_[KERNEL, HEAP, ARG0..ARG2];
  print "SERVER: recieved a connection from $peeraddr:$peerport\n";
```

```
  POE::Session->create(
    inline_states => {
      _start      => \&start_child,
      socket_input => \&socket_input,
      socket_failed => \&socket_failed,
      check_shutdown => \&check_shutdown,
    },
    args => [ $socket, $peeraddr, $peerport ],
  );
}
```

Create a session for each connection

```
sub start_child {
  my ($heap, $socket, $peeraddr, $peerport) = @_[HEAP, ARG0..ARG2];
```

```
  $heap->{readwrite} = POE::Wheel::ReadWrite->new(
    Handle => $socket,
    Driver => POE::Driver::SysRW->new(),
    Filter => POE::Filter::Line->new(),
    InputEvent => 'socket_input',
    ErrorEvent => 'socket_failed',
    FlushedEvent => 'check_shutdown',
  );
}
```

Use a ReadWrite Wheel for the connection

```
sub check_shutdown {
  delete $_[HEAP]->{readwrite} if ($_[HEAP]->{shutdown_now});
}
```

```
sub socket_input {
  my ($heap, $buf) = @_[HEAP, ARG0];
  $heap->{readwrite}->put($buf);
  $heap->{shutdown_now} = 1 if ($buf eq "QUIT");
}
```

Echo the data back over the socket

```
sub socket_failed {
  delete $_[HEAP]->{readwrite};
}
```

# Socket Example (cont.)

```
sub accept_client {
  my ($kernel, $heap, $socket, $peeraddr, $peerport) = @_[KERNEL, HEAP, ARG0..ARG2];
  print "SERVER: recieved a connection from $peeraddr:$peerport\n";
```

```
  POE::Session->create(
    inline_states => {
      _start      => \&start_child,
      socket_input => \&socket_input,
      socket_failed => \&socket_failed,
      check_shutdown => \&check_shutdown,
    },
    args => [ $socket, $peeraddr, $peerport ],
  );
}
```

Create a session for each connection

```
sub start_child {
  my ($heap, $socket, $peeraddr, $peerport) = @_[HEAP, ARG0..ARG2];
```

```
  $heap->{readwrite} = POE::Wheel::ReadWrite->new(
    Handle => $socket,
    Driver => POE::Driver::SysRW->new(),
    Filter => POE::Filter::Line->new(),
    InputEvent => 'socket_input',
    ErrorEvent => 'socket_failed',
    FlushedEvent => 'check_shutdown',
  );
}
```

Use a ReadWrite Wheel for the connection

```
sub check_shutdown {
  delete $_[HEAP]->{readwrite} if ($_[HEAP]->{shutdown_now});
}
```

```
sub socket_input {
  my ($heap, $buf) = @_[HEAP, ARG0];
  $heap->{readwrite}->put($buf);
  $heap->{shutdown_now} = 1 if ($buf eq "QUIT");
}
```

Echo the data back over the socket

```
sub socket_failed {
  delete $_[HEAP]->{readwrite};
}
```





# Wheel Drivers and Filters



# Wheel Drivers and Filters

```
POE::Wheel::ReadWrite->new(  
    Handle => $socket,  
    Driver => POE::Driver::SysRW->new(),  
    Filter => POE::Filter::Line->new(),  
    InputEvent => 'socket_input',  
    ErrorEvent => 'socket_failed',  
    FlushedEvent => 'check_shutdown',  
);
```

# Wheel Drivers and Filters

```
POE::Wheel::ReadWrite->new(  
    Handle => $socket,  
    Driver => POE::Driver::SysRW->new(),  
    Filter => POE::Filter::Line->new(),  
    InputEvent => 'socket_input',  
    ErrorEvent => 'socket_failed',  
    FlushedEvent => 'check_shutdown',  
);
```

- *Drivers* are what feeds the wheel. Only SysRW exists right now

# Wheel Drivers and Filters

```
POE::Wheel::ReadWrite->new(  
    Handle => $socket,  
    Driver => POE::Driver::SysRW->new(),  
    Filter => POE::Filter::Line->new(),  
    InputEvent => 'socket_input',  
    ErrorEvent => 'socket_failed',  
    FlushedEvent => 'check_shutdown',  
);
```

- *Drivers* are what feeds the wheel. Only SysRW exists right now
- *Filters* bundle up what comes out of the Driver into usable chunks

# Wheel Drivers and Filters

```
POE::Wheel::ReadWrite->new(  
  Handle => $socket,  
  Driver => POE::Driver::SysRW->new(),  
  Filter => POE::Filter::Line->new(),  
  InputEvent => 'socket_input',  
  ErrorEvent => 'socket_failed',  
  FlushedEvent => 'check_shutdown',  
);
```

- *Drivers* are what feeds the wheel. Only SysRW exists right now
- *Filters* bundle up what comes out of the Driver into usable chunks
  - ◆ e.g. Filter::Line bundles into lines

# Wheel Drivers and Filters

```
POE::Wheel::ReadWrite->new(  
  Handle => $socket,  
  Driver => POE::Driver::SysRW->new(),  
  Filter => POE::Filter::Line->new(),  
  InputEvent => 'socket_input',  
  ErrorEvent => 'socket_failed',  
  FlushedEvent => 'check_shutdown',  
);
```

- *Drivers* are what feeds the wheel. Only SysRW exists right now
- *Filters* bundle up what comes out of the Driver into usable chunks
  - ◆ e.g. Filter::Line bundles into lines
- Can have different filters for Input and Output (via the InputFilter and OutputFilter parameters)

# Wheel Drivers and Filters

```
POE::Wheel::ReadWrite->new(  
  Handle => $socket,  
  Driver => POE::Driver::SysRW->new(),  
  Filter => POE::Filter::Line->new(),  
  InputEvent => 'socket_input',  
  ErrorEvent => 'socket_failed',  
  FlushedEvent => 'check_shutdown',  
);
```

- *Drivers* are what feeds the wheel. Only SysRW exists right now
- *Filters* bundle up what comes out of the Driver into usable chunks
  - ◆ e.g. Filter::Line bundles into lines
- Can have different filters for Input and Output (via the InputFilter and OutputFilter parameters)
- Filters can return objects too - e.g. POE::Filter::HTTPD



# Socket Example - Graceful Shutdown





# Socket Example - Graceful Shutdown

- When data comes in we check its contents:

# Socket Example - Graceful Shutdown

- When data comes in we check its contents:

```
sub socket_input {  
  my ($heap, $buf) = @_[HEAP, ARG0];  
  $heap->{readwrite}->put($buf);  
  $heap->{shutdown_now} = 1 if ($buf eq "QUIT");  
}
```

# Socket Example - Graceful Shutdown

- When data comes in we check its contents:

```
sub socket_input {  
  my ($heap, $buf) = @_[HEAP, ARG0];  
  $heap->{readwrite}->put($buf);  
  $heap->{shutdown_now} = 1 if ($buf eq "QUIT");  
}
```

- ... and remember that we have to shut down

# Socket Example - Graceful Shutdown

- When data comes in we check its contents:

```
sub socket_input {  
  my ($heap, $buf) = @_[HEAP, ARG0];  
  $heap->{readwrite}->put($buf);  
  $heap->{shutdown_now} = 1 if ($buf eq "QUIT");  
}
```

- ... and remember that we have to shut down
- If we shut down immediately the client wouldn't see the last bit of data due to buffering

# Socket Example - Graceful Shutdown

- When data comes in we check its contents:

```
sub socket_input {  
  my ($heap, $buf) = @_[HEAP, ARG0];  
  $heap->{readwrite}->put($buf);  
  $heap->{shutdown_now} = 1 if ($buf eq "QUIT");  
}
```

- ... and remember that we have to shut down
- If we shut down immediately the client wouldn't see the last bit of data due to buffering
- Instead we shut down when the data is flushed:

# Socket Example - Graceful Shutdown

- When data comes in we check its contents:

```
sub socket_input {  
  my ($heap, $buf) = @_[HEAP, ARG0];  
  $heap->{readwrite}->put($buf);  
  $heap->{shutdown_now} = 1 if ($buf eq "QUIT");  
}
```

- ... and remember that we have to shut down
- If we shut down immediately the client wouldn't see the last bit of data due to buffering
- Instead we shut down when the data is flushed:

```
    FlushedEvent => 'check_shutdown',  
  );  
}  
  
sub check_shutdown {  
  delete $_[HEAP]->{readwrite} if ($_[HEAP]->{shutdown_now});  
}
```

# Socket Example - Graceful Shutdown

- When data comes in we check its contents:

```
sub socket_input {  
  my ($heap, $buf) = @_[HEAP, ARG0];  
  $heap->{readwrite}->put($buf);  
  $heap->{shutdown_now} = 1 if ($buf eq "QUIT");  
}
```

- ... and remember that we have to shut down
- If we shut down immediately the client wouldn't see the last bit of data due to buffering
- Instead we shut down when the data is flushed:

```
    FlushedEvent => 'check_shutdown',  
  );  
}  
  
sub check_shutdown {  
  delete $_[HEAP]->{readwrite} if ($_[HEAP]->{shutdown_now});  
}
```

- Shutdown is achieved by deleting the wheel



# Socket Example - Simplify With Abstraction



# Socket Example - Simplify With Abstraction

```
use POE;
use POE::Component::Server::TCP;

POE::Session->create(
  inline_states => {
    _start => sub {
      $_[HEAP]->{server} = POE::Component::Server::TCP->new(
        Port          => 5555,
        Address       => '127.0.0.1',
        ClientInput   => sub { $_[HEAP]->{client}->put($_[ARG0]) },
        ClientFilter  => POE::Filter::Line->new(),
      );
    },
  },
);

$poe_kernel->run();
exit(0);
```

# Socket Example - Simplify With Abstraction

```
use POE;  
use POE::Component::Server::TCP;  
  
POE::Session->create(  
  inline_states => {  
    _start => sub {  
      $_[HEAP]->{server} = POE::Component::Server::TCP->new(  
        Port          => 5555,  
        Address       => '127.0.0.1',  
        ClientInput   => sub { $_[HEAP]->{client}->put($_[ARG0]) },  
        ClientFilter  => POE::Filter::Line->new(),  
      );  
    },  
  },  
);  
  
$poe_kernel->run();  
exit(0);
```

- Much simpler!

# Socket Example - Simplify With Abstraction

```
use POE;  
use POE::Component::Server::TCP;  
  
POE::Session->create(  
  inline_states => {  
    _start => sub {  
      $_[HEAP]->{server} = POE::Component::Server::TCP->new(  
        Port          => 5555,  
        Address       => '127.0.0.1',  
        ClientInput   => sub { $_[HEAP]->{client}->put($_[ARG0]) },  
        ClientFilter  => POE::Filter::Line->new(),  
      );  
    },  
  },  
);  
  
$poe_kernel->run();  
exit(0);
```

- Much simpler!
  - ◆ But now at least you know what POE::Component::Server::TCP is doing internally

# Socket Example - Simplify With Abstraction

```
use POE;  
use POE::Component::Server::TCP;  
  
POE::Session->create(  
  inline_states => {  
    _start => sub {  
      $_[HEAP]->{server} = POE::Component::Server::TCP->new(  
        Port          => 5555,  
        Address       => '127.0.0.1',  
        ClientInput   => sub { $_[HEAP]->{client}->put($_[ARG0]) },  
        ClientFilter  => POE::Filter::Line->new(),  
      );  
    },  
  },  
);  
  
$poe_kernel->run();  
exit(0);
```

- Much simpler!
  - ◆ But now at least you know what POE::Component::Server::TCP is doing internally
- Note that this one doesn't do graceful shutdown. That's left as an exercise :-)

# Socket Example - Simplify With Abstraction

```
use POE;  
use POE::Component::Server::TCP;  
  
POE::Session->create(  
  inline_states => {  
    _start => sub {  
      $_[HEAP]->{server} = POE::Component::Server::TCP->new(  
        Port          => 5555,  
        Address       => '127.0.0.1',  
        ClientInput   => sub { $_[HEAP]->{client}->put($_[ARG0]) },  
        ClientFilter  => POE::Filter::Line->new(),  
      );  
    },  
  },  
);  
  
$poe_kernel->run();  
exit(0);
```

- Much simpler!
  - ◆ But now at least you know what POE::Component::Server::TCP is doing internally
- Note that this one doesn't do graceful shutdown. That's left as an exercise :-)





# Wheels Roundup





# Wheels Roundup

- Wheels are one way a session can receive events

# Wheels Roundup

- Wheels are one way a session can receive events
- POE's Wheel abstractions make non-blocking I/O easy



# Wheels Roundup

- Wheels are one way a session can receive events
- POE's Wheel abstractions make non-blocking I/O easy
- Other Wheels are available:

# Wheels Roundup

- Wheels are one way a session can receive events
- POE's Wheel abstractions make non-blocking I/O easy
- Other Wheels are available:
  - ◆ `POE::Wheel::Curses` - full screen console I/O

# Wheels Roundup

- Wheels are one way a session can receive events
- POE's Wheel abstractions make non-blocking I/O easy
- Other Wheels are available:
  - ◆ POE::Wheel::Curses - full screen console I/O
  - ◆ POE::Wheel::ListenAccept - I/O on existing sockets

# Wheels Roundup

- Wheels are one way a session can receive events
- POE's Wheel abstractions make non-blocking I/O easy
- Other Wheels are available:
  - ◆ POE::Wheel::Curses - full screen console I/O
  - ◆ POE::Wheel::ListenAccept - I/O on existing sockets
  - ◆ POE::Wheel::ReadLine - Input from the command line

# Wheels Roundup

- Wheels are one way a session can receive events
- POE's Wheel abstractions make non-blocking I/O easy
- Other Wheels are available:
  - ◆ POE::Wheel::Curses - full screen console I/O
  - ◆ POE::Wheel::ListenAccept - I/O on existing sockets
  - ◆ POE::Wheel::ReadLine - Input from the command line
    - i.e. like my \$line = <STDIN>; but without the blocking

# Wheels Roundup

- Wheels are one way a session can receive events
- POE's Wheel abstractions make non-blocking I/O easy
- Other Wheels are available:
  - ◆ POE::Wheel::Curses - full screen console I/O
  - ◆ POE::Wheel::ListenAccept - I/O on existing sockets
  - ◆ POE::Wheel::ReadLine - Input from the command line
    - i.e. like my \$line = <STDIN>; but without the blocking
  - ◆ POE::Wheel::Run - I/O to and from subprocesses

# **Advanced POE - How the Kernel Works**



# POE's Event Queue



# POE's Event Queue

- A Priority Queue

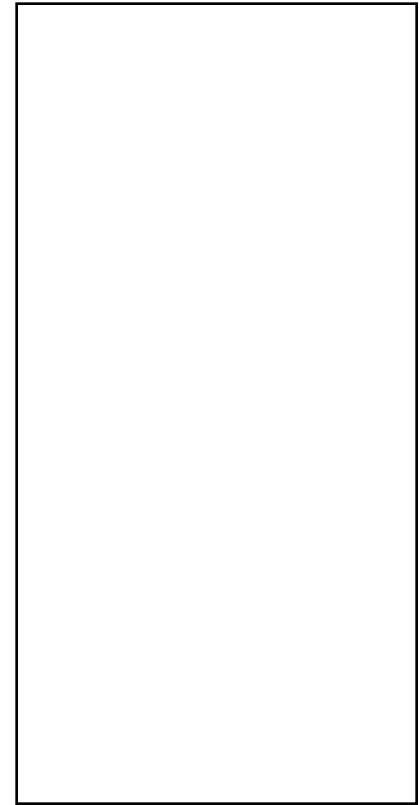


# POE's Event Queue

- A Priority Queue
- Priority is based on epoch time

# POE's Event Queue

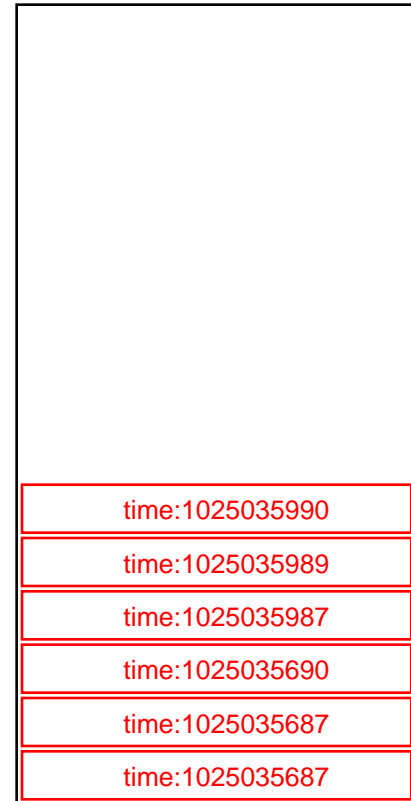
- A Priority Queue
- Priority is based on epoch time



The Queue

# POE's Event Queue

- A Priority Queue
- Priority is based on epoch time



Events in the Queue



The Queue

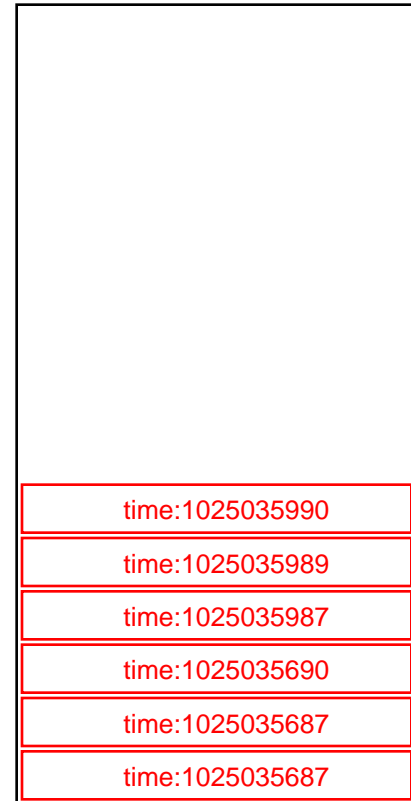
Current time: 1025035687

# POE's Event Queue

- A Priority Queue
- Priority is based on epoch time

A new event, via `$kernel->yield()`

time:1025035687



Events in the Queue



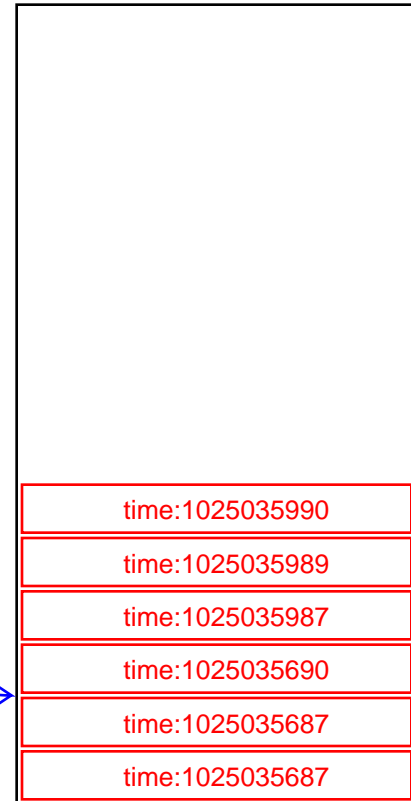
The Queue

# POE's Event Queue

- A Priority Queue
- Priority is based on epoch time

A new event, via `$kernel->yield()`

time:1025035687



Current time: 1025035687

Events in the Queue

The Queue

# POE's Event Queue

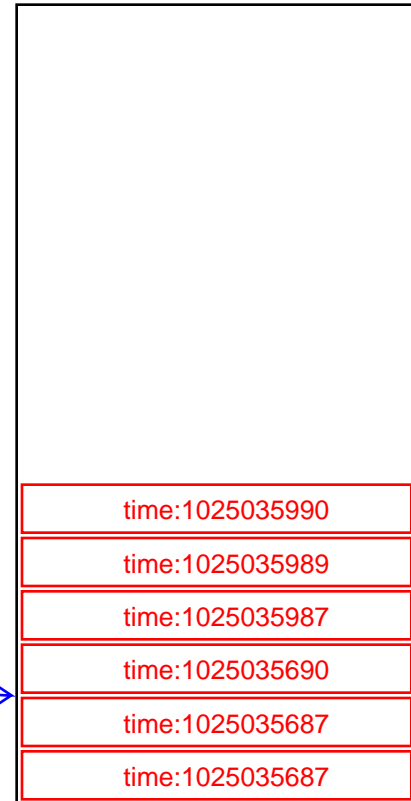
- A Priority Queue
- Priority is based on epoch time

A new event, via `$kernel->yield()`

time:1025035687

A new event, via  
`$kernel->alarm_set("event", time + 3*60)`

time:1025035867



Events in the Queue

The Queue

# POE's Event Queue

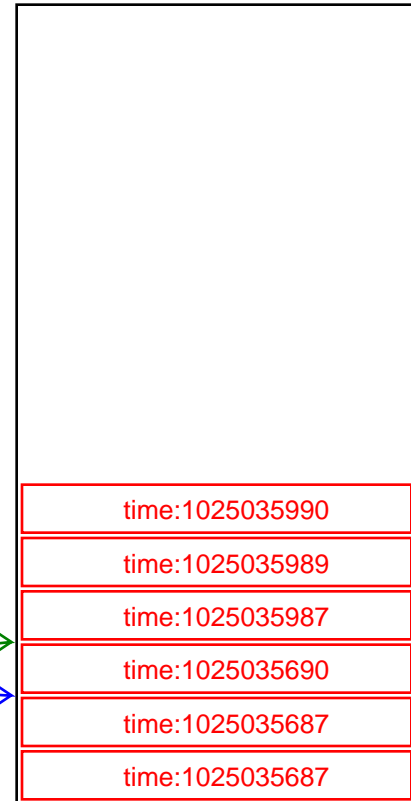
- A Priority Queue
- Priority is based on epoch time

A new event, via `$kernel->yield()`

time:1025035687

A new event, via  
`$kernel->alarm_set("event", time + 3*60)`

time:1025035867



Events in the Queue

The Queue

# POE's Event Loop

- Wait for I/O

```
# Wait timeout:  
my $timeout = $queue[0]->[TIME] - time();
```

- Enqueue I/O events
- Dispatch events until topmost event time > \$now
  - ◆ After each returns, do garbage collection
- Alternate event loops (e.g. Gtk, Tk) just wait for different kinds of I/O events
- Exit the loop when there are no more sessions left



# Kernel Multitasking



# Kernel Multitasking

- POE "Multitasks", but it doesn't `fork()` or `thread->new()`



# Kernel Multitasking

- POE "Multitasks", but it doesn't `fork()` or `thread->new()`
- Each "Wait for I/O" only waits until the next event in the queue is due



# Kernel Multitasking

- POE "Multitasks", but it doesn't fork() or thread->new()
- Each "Wait for I/O" only waits until the next event in the queue is due
- This means all events get scheduled in a timely manner



# Kernel Multitasking

- POE "Multitasks", but it doesn't fork() or thread->new()
- Each "Wait for I/O" only waits until the next event in the queue is due
- This means all events get scheduled in a timely manner
- The *OS Kernel* lets I/O happen in the background



# Kernel Multitasking

- POE "Multitasks", but it doesn't fork() or thread->new()
- Each "Wait for I/O" only waits until the next event in the queue is due
- This means all events get scheduled in a timely manner
- The *OS Kernel* lets I/O happen in the background
- When running with Tk or Gtk, the GUI events can carry on in the background



# **Advanced POE - Keeping POE Alive**

**How to avoid the grim reaper**



# When POE Exits



# When POE Exits

- When it has nothing to do!

# When POE Exits

- When it has nothing to do!
- What constitutes something to do?

# When POE Exits

- When it has nothing to do!
- What constitutes something to do?
  - ◆ A pending event in the queue

# When POE Exits

- When it has nothing to do!
- What constitutes something to do?
  - ◆ A pending event in the queue
  - ◆ An active *Wheel* waiting for something to happen

# When POE Exits

- When it has nothing to do!
- What constitutes something to do?
  - ◆ A pending event in the queue
  - ◆ An active *Wheel* waiting for something to happen
  - ◆ Alive sessions waiting on the event queue

# When POE Exits

- When it has nothing to do!
- What constitutes something to do?
  - ◆ A pending event in the queue
  - ◆ An active *Wheel* waiting for something to happen
  - ◆ Alive sessions waiting on the event queue
- If POE exits and you didn't expect it to, you probably forgot something

# When POE Exits

- When it has nothing to do!
- What constitutes something to do?
  - ◆ A pending event in the queue
  - ◆ An active Wheel waiting for something to happen
  - ◆ Alive sessions waiting on the event queue
- If POE exits and you didn't expect it to, you probably forgot something

```
use POE;
$|++;

POE::Session->create(
    inline_states => {
        _start => sub { $_[KERNEL]->yield('do_something') }
    });

$poe_kernel->run();
exit(0);

sub do_something {
    my ($kernel, $heap) = @_[KERNEL, HEAP];
    print "This sub is supposed to do something, then loop\n";
    $kernel->yield('do_something');
}
```

# When POE Exits

- When it has nothing to do!
- What constitutes something to do?
  - ◆ A pending event in the queue
  - ◆ An active Wheel waiting for something to happen
  - ◆ Alive sessions waiting on the event queue
- If POE exits and you didn't expect it to, you probably forgot something

```
use POE;  
$|++;
```

```
POE::Session->create(  
  inline_states => {  
    _start => sub { $_[KERNEL]->yield('do_something') }  
  });
```

Oops, we forgot to define the "do\_something" state

```
$poe_kernel->run();  
exit(0);
```

```
sub do_something {  
  my ($kernel, $heap) = @_[KERNEL, HEAP];  
  print "This sub is supposed to do something, then loop\n";  
  $kernel->yield('do_something');  
}
```

# When POE Exits

- When it has nothing to do!
- What constitutes something to do?
  - ◆ A pending event in the queue
  - ◆ An active Wheel waiting for something to happen
  - ◆ Alive sessions waiting on the event queue
- If POE exits and you didn't expect it to, you probably forgot something

```
use POE;  
$|++;
```

```
POE::Session->create(  
  inline_states => {  
    _start => sub { $_[KERNEL]->yield('do_something') }  
  });
```

Oops, we forgot to define the "do\_something" state

```
$poe_kernel->run();  
exit(0);
```

```
sub do_something {  
  my ($kernel, $heap) = @_[KERNEL, HEAP];  
  print "This sub is supposed to do something, then loop\n";  
  $kernel->yield('do_something');  
}
```





# Real World Example - File Processor



# Real World Example - File Processor

```
use POE;

POE::Session->create(
    inline_states => {
        _start => \&process_file,
        process_file => \&process_file,
    }
    args => [ @ARGV ],
);

$poe_kernel->run();
exit(0);

sub process_file {
    my ($kernel, @dirs) = @_ [KERNEL, ARG0..$#_];

    my $file;

    DIRECTORY:
    foreach my $dir (@dirs) {
        opendir(DIR, $dir) || die "Can't open dir $dir: $!";
        while (my $filename = readdir(DIR)) {
            $filename = "$dir/$filename";
            if (-f $filename) {
                $file = $filename;
                last DIRECTORY;
            }
        }
        closedir(DIR);
    }

    if ($file) {
        do_something_with_file($file); # this deletes $file
        $kernel->yield('process_file', @dirs);
    }
    # hint: we forgot something here
}
```

# Real World Example - File Processor

```
use POE;

POE::Session->create(
    inline_states => {
        _start => \&process_file,
        process_file => \&process_file,
    }
    args => [ @ARGV ],
);

$poe_kernel->run();
exit(0);

sub process_file {
    my ($kernel, @dirs) = @_ [KERNEL, ARG0..$#_];

    my $file;

    DIRECTORY:
    foreach my $dir (@dirs) {
        opendir(DIR, $dir) || die "Can't open dir $dir: $!";
        while (my $filename = readdir(DIR)) {
            $filename = "$dir/$filename";
            if (-f $filename) {
                $file = $filename;
                last DIRECTORY;
            }
        }
        closedir(DIR);
    }

    if ($file) {
        do_something_with_file($file); # this deletes $file
        $kernel->yield('process_file', @dirs);
    }
    # hint: we forgot something here
}
```





# IRC Bots





# IRC Bots

- What happens with network errors?

# IRC Bots

- What happens with network errors?

```
sub irc_disconnect {
    my ($kernel, $heap, $server) = @_[KERNEL, HEAP, ARG0];
    $heap->{irc_connected} = 0;
    $kernel->yield('reconnect', $server);
}
```

```
sub reconnect {
    my ($kernel, $heap) = @_[KERNEL, HEAP];
    return if $heap->{irc_connected};

    my $opt = $heap->{options};

    $kernel->post( $opt->{bot_id}, 'connect',
        {
            Nick => $options->{nick},
            Server => $options->{server},
            Port => $options->{port} || 6667,
            Username => $options->{user},
            Ircname => "Blog Bot",
        },
    );

    # try again in 2 seconds (may have succeeded, but we test that at the top
    $kernel->delay_set( 'reconnect', 2 );
}

sub irc_connected {
    $_[HEAP]->{irc_connected} = 1;
}
```



# Daemonizing POE





# Daemonizing POE

- We often write daemons with POE (network daemons, cron daemons, etc)

# Daemonizing POE

- We often write daemons with POE (network daemons, cron daemons, etc)
- But POE doesn't "background" automatically, so we need to write that:

# Daemonizing POE

- We often write daemons with POE (network daemons, cron daemons, etc)
- But POE doesn't "background" automatically, so we need to write that:

```
use POE;
use POSIX;
$|++;

sub become_daemon {
    my $child = fork;
    die "Can't fork: $!" unless defined($child);
    exit(0) if $child;    # parent dies;
    POSIX::setsid();    # become session leader
    open(STDIN, "</dev/null");
    open(STDOUT, ">/dev/null");
    open(STDERR, '>&STDOUT');
    umask(0);            # forget file mode creation mask
    $ENV{PATH} = '/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin';
    delete @ENV{'IFS', 'CDPATH', 'ENV', 'BASH_ENV'};
}

become_daemon(); # MUST do this before we create any sessions

POE::Session->create(
    inline_states => {
        _start => sub { $_[KERNEL]->yield("loop") },
        loop => sub { $_[KERNEL]->delay_set("loop", 10) },
    });

$poe_kernel->run();
exit(0);
```

# Daemonizing POE

- We often write daemons with POE (network daemons, cron daemons, etc)
- But POE doesn't "background" automatically, so we need to write that:

```
use POE;
use POSIX;
$|++;

sub become_daemon {
    my $child = fork;
    die "Can't fork: $!" unless defined($child);
    exit(0) if $child;    # parent dies;
    POSIX::setsid();    # become session leader
    open(STDIN, "</dev/null");
    open(STDOUT, ">/dev/null");
    open(STDERR, '>&STDOUT');
    umask(0);            # forget file mode creation mask
    $ENV{PATH} = '/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin';
    delete @ENV{'IFS', 'CDPATH', 'ENV', 'BASH_ENV'};
}

become_daemon(); # MUST do this before we create any sessions

POE::Session->create(
    inline_states => {
        _start => sub { $_[KERNEL]->yield("loop") },
        loop => sub { $_[KERNEL]->delay_set("loop", 10) },
    });

$poe_kernel->run();
exit(0);
```





# **Advanced POE - Sessions in Depth**



# Unhandled Events





# Unhandled Events

- As we saw - an unhandled event is silently ignored



# Unhandled Events

- As we saw - an unhandled event is silently ignored
- We can "fix" this with POE's `ASSERT_DEFAULT` constant:

# Unhandled Events

- As we saw - an unhandled event is silently ignored
- We can "fix" this with POE's ASSERT\_DEFAULT constant:

```
sub POE::Kernel::ASSERT_DEFAULT () { 1 }
use POE;
$|++;

POE::Session->create(
    inline_states => {
        _start => sub { $_[KERNEL]->yield('do_something') }
    });

$poe_kernel->run();
exit(0);

sub do_something {
    my ($kernel, $heap) = @_[KERNEL, HEAP];
    print "This sub is supposed to do something, then loop\n";
    $kernel->yield('do_something');
}
```

# Unhandled Events

- As we saw - an unhandled event is silently ignored
- We can "fix" this with POE's ASSERT\_DEFAULT constant:

```
sub POE::Kernel::ASSERT_DEFAULT () { 1 }
use POE;
$|++;

POE::Session->create(
    inline_states => {
        _start => sub { $_[KERNEL]->yield('do_something') }
    });

$poe_kernel->run();
exit(0);

sub do_something {
    my ($kernel, $heap) = @_[KERNEL, HEAP];
    print "This sub is supposed to do something, then loop\n";
    $kernel->yield('do_something');
}
```





# Unhandled Events (cont.)





# Unhandled Events (cont.)

- Alternatively we can implement the `_default` handler

# Unhandled Events (cont.)

- Alternatively we can implement the `_default` handler

```
use POE;
$|++;

POE::Session->create(
    inline_states => {
        _start => sub { $_[KERNEL]->yield('do_something') },
        _default => \&default_handler,
    });

$poe_kernel->run();
exit(0);

sub default_handler {
    warn("The $_[ARG0] event was called but didn't exist.\n");
    warn("Params: ", join(', ', @{$_[ARG1]}), "\n");
    return 0; # otherwise we mess up signal handling
}

sub do_something {
    my ($kernel, $heap) = @_[KERNEL, HEAP];
    print "This sub is supposed to do something, then loop\n";
    $kernel->yield('do_something');
}
```

# Unhandled Events (cont.)

- Alternatively we can implement the `_default` handler

```
use POE;
$|++;

POE::Session->create(
    inline_states => {
        _start => sub { $_[KERNEL]->yield('do_something') },
        _default => \&default_handler,
    });

$poe_kernel->run();
exit(0);

sub default_handler {
    warn("The $_[ARG0] event was called but didn't exist.\n");
    warn("Params: ", join(', ', @{$_[ARG1]}), "\n");
    return 0; # otherwise we mess up signal handling
}

sub do_something {
    my ($kernel, $heap) = @_[KERNEL, HEAP];
    print "This sub is supposed to do something, then loop\n";
    $kernel->yield('do_something');
}
```





# Session Construction





# Session Construction

- Sessions support three different types of states



# Session Construction

- Sessions support three different types of states
  - ◆ Inline states (as we have seen so far)



# Session Construction

- Sessions support three different types of states
  - ◆ Inline states (as we have seen so far)
  - ◆ Package states



# Session Construction

- Sessions support three different types of states
  - ◆ Inline states (as we have seen so far)
  - ◆ Package states
  - ◆ Object states



# Session Construction

- Sessions support three different types of states
  - ◆ Inline states (as we have seen so far)
  - ◆ Package states
  - ◆ Object states
- Inline states are simply functions

# Session Construction

- Sessions support three different types of states
  - ◆ Inline states (as we have seen so far)
  - ◆ Package states
  - ◆ Object states
- Inline states are simply functions
- Object states use OO method calls, and pass the object in as `$_[OBJECT]`



# Session Construction

- Sessions support three different types of states
  - ◆ Inline states (as we have seen so far)
  - ◆ Package states
  - ◆ Object states
- Inline states are simply functions
- Object states use OO method calls, and pass the object in as `$_[OBJECT]`
- Package states use class method calls, and pass the class in as `$_[OBJECT]`



# Object States



# Object States

```
use POE;
$|++;

my $obj = SubClass->new();

POE::Session->create(
    inline_states => {
        _start => sub { $_[KERNEL]->yield('spurt') },
    },
    object_states => [
        $obj => [ 'gurgle', 'spurt' ],
    ],
);
$poe_kernel->run();
exit(0);

package BaseClass;
use POE; # needed to import the KERNEL and OBJECT constants

sub new {
    my $class = shift;
    return bless {}, $class;
}
sub spurt {
    my ($object, $kernel) = @_[OBJECT, KERNEL];
    print "I'm spurting in the base class first!\n";
    $kernel->yield('gurgle');
}

package SubClass;
BEGIN { @ISA = qw(BaseClass); }
use POE; # needed to import the KERNEL and OBJECT constants

sub gurgle {
    my ($object, $kernel) = @_[OBJECT, KERNEL];
    print "And now I'm gurgling in the subclass\n";
}
```

# Object States

```
use POE;
$|++;

my $obj = SubClass->new();

POE::Session->create(
    inline_states => {
        _start => sub { $_[KERNEL]->yield('spurt') },
    },
    object_states => [
        $obj => [ 'gurgle', 'spurt' ],
    ],
);
$poe_kernel->run();
exit(0);

package BaseClass;
use POE; # needed to import the KERNEL and OBJECT constants

sub new {
    my $class = shift;
    return bless {}, $class;
}
sub spurt {
    my ($object, $kernel) = @_[OBJECT, KERNEL];
    print "I'm spurting in the base class first!\n";
    $kernel->yield('gurgle');
}

package SubClass;
BEGIN { @ISA = qw(BaseClass); }
use POE; # needed to import the KERNEL and OBJECT constants

sub gurgle {
    my ($object, $kernel) = @_[OBJECT, KERNEL];
    print "And now I'm gurgling in the subclass\n";
}
```





# Session Options



# Session Options

- POE::Session->create(options => { ... });

# Session Options

- POE::Session->create(options => { ... });
- Two options: *trace* and *debug*

# Session Options

- POE::Session->create(options => { ... });
- Two options: *trace* and *debug*
- ***trace*** => 1

# Session Options

- POE::Session->create(options => { ... });
- Two options: *trace* and *debug*
- ***trace*** => 1
  - ◆ Traces all invocations on this session

# Session Options

- POE::Session->create(options => { ... });
- Two options: *trace* and *debug*
- ***trace*** => 1
  - ◆ Traces all invocations on this session
  - ◆ Useful for watching what happens to this session without having to watch all other sessions

# Session Options

- POE::Session->create(options => { ... });
- Two options: *trace* and *debug*
- ***trace*** => 1
  - ◆ Traces all invocations on this session
  - ◆ Useful for watching what happens to this session without having to watch all other sessions
- ***debug*** => 1

# Session Options

- POE::Session->create(options => { ... });
- Two options: *trace* and *debug*
- ***trace*** => 1
  - ◆ Traces all invocations on this session
  - ◆ Useful for watching what happens to this session without having to watch all other sessions
- ***debug*** => 1
  - ◆ Same as ASSERT\_DEFAULT, but for a single session



# Other Event Parameters



# Other Event Parameters

- As we have seen, events receive lots of parameters



# Other Event Parameters

- As we have seen, events receive lots of parameters
- KERNEL, HEAP, SESSION, OBJECT, ARG0..\$#\_



# Other Event Parameters

- As we have seen, events receive lots of parameters
- KERNEL, HEAP, SESSION, OBJECT, ARG0..\$#\_
- But also: SENDER, STATE, CALLER\_LINE, CALLER\_FILE



# Other Event Parameters

- As we have seen, events receive lots of parameters
- KERNEL, HEAP, SESSION, OBJECT, ARG0..\$#\_
- But also: SENDER, STATE, CALLER\_LINE, CALLER\_FILE
- SENDER

# Other Event Parameters

- As we have seen, events receive lots of parameters
- KERNEL, HEAP, SESSION, OBJECT, ARG0..\$#\_
- But also: SENDER, STATE, CALLER\_LINE, CALLER\_FILE
- SENDER
  - ◆ The session that caused this state to be invoked

# Other Event Parameters

- As we have seen, events receive lots of parameters
- KERNEL, HEAP, SESSION, OBJECT, ARG0..\$\_#\_
- But also: SENDER, STATE, CALLER\_LINE, CALLER\_FILE
- SENDER
  - ◆ The session that caused this state to be invoked
  - ◆ Use in callbacks, for a "reply-to"

# Other Event Parameters

- As we have seen, events receive lots of parameters
- KERNEL, HEAP, SESSION, OBJECT, ARG0..\$#\_
- But also: SENDER, STATE, CALLER\_LINE, CALLER\_FILE
- SENDER
  - ◆ The session that caused this state to be invoked
  - ◆ Use in callbacks, for a "reply-to"
- STATE

# Other Event Parameters

- As we have seen, events receive lots of parameters
- KERNEL, HEAP, SESSION, OBJECT, ARG0..\$#\_
- But also: SENDER, STATE, CALLER\_LINE, CALLER\_FILE
- SENDER
  - ◆ The session that caused this state to be invoked
  - ◆ Use in callbacks, for a "reply-to"
- STATE
  - ◆ The name of this state

# Other Event Parameters

- As we have seen, events receive lots of parameters
- KERNEL, HEAP, SESSION, OBJECT, ARG0..\$#\_
- But also: SENDER, STATE, CALLER\_LINE, CALLER\_FILE
- SENDER
  - ◆ The session that caused this state to be invoked
  - ◆ Use in callbacks, for a "reply-to"
- STATE
  - ◆ The name of this state
  - ◆ Use for *\_default* to determine the name of the state that was supposed to be invoked

# Other Event Parameters

- As we have seen, events receive lots of parameters
- KERNEL, HEAP, SESSION, OBJECT, ARG0..\$#\_
- But also: SENDER, STATE, CALLER\_LINE, CALLER\_FILE
- SENDER
  - ◆ The session that caused this state to be invoked
  - ◆ Use in callbacks, for a "reply-to"
- STATE
  - ◆ The name of this state
  - ◆ Use for *\_default* to determine the name of the state that was supposed to be invoked
- CALLER\_LINE and CALLER\_FILE

# Other Event Parameters

- As we have seen, events receive lots of parameters
- KERNEL, HEAP, SESSION, OBJECT, ARG0..\$#\_
- But also: SENDER, STATE, CALLER\_LINE, CALLER\_FILE
- SENDER
  - ◆ The session that caused this state to be invoked
  - ◆ Use in callbacks, for a "reply-to"
- STATE
  - ◆ The name of this state
  - ◆ Use for *\_default* to determine the name of the state that was supposed to be invoked
- CALLER\_LINE and CALLER\_FILE
  - ◆ The line and filename of where this event got called from

# Other Event Parameters

- As we have seen, events receive lots of parameters
- KERNEL, HEAP, SESSION, OBJECT, ARG0..\$#\_
- But also: SENDER, STATE, CALLER\_LINE, CALLER\_FILE
- SENDER
  - ◆ The session that caused this state to be invoked
  - ◆ Use in callbacks, for a "reply-to"
- STATE
  - ◆ The name of this state
  - ◆ Use for *\_default* to determine the name of the state that was supposed to be invoked
- CALLER\_LINE and CALLER\_FILE
  - ◆ The line and filename of where this event got called from
  - ◆ Again, useful in *\_default* for debugging



# Session Pre-defined Events





# Session Pre-defined Events

- `_start/_stop`



# Session Pre-defined Events

- `_start/_stop`
  - ◆ Called when the session starts and stops

# Session Pre-defined Events

- `_start/_stop`
  - ◆ Called when the session starts and stops
- `_child`



# Session Pre-defined Events

- `_start/_stop`
  - ◆ Called when the session starts and stops
- `_child`
  - ◆ Called when a child session is created or dies



# Session Pre-defined Events

- `_start/_stop`
  - ◆ Called when the session starts and stops
- `_child`
  - ◆ Called when a child session is created or dies
  - ◆ ARG0 contains "gain", "lose", or "create" depending on what actually happened



# Session Pre-defined Events

- `_start/_stop`
  - ◆ Called when the session starts and stops
- `_child`
  - ◆ Called when a child session is created or dies
  - ◆ ARG0 contains "gain", "lose", or "create" depending on what actually happened
  - ◆ ARG1 contains a reference to the session

# Session Pre-defined Events

- `_start/_stop`
  - ◆ Called when the session starts and stops
- `_child`
  - ◆ Called when a child session is created or dies
  - ◆ ARG0 contains "gain", "lose", or "create" depending on what actually happened
  - ◆ ARG1 contains a reference to the session
  - ◆ When ARG0 is "create", ARG2 contains the return value from the session's `_start` state

# Session Pre-defined Events

- `_start/_stop`
  - ◆ Called when the session starts and stops
- `_child`
  - ◆ Called when a child session is created or dies
  - ◆ ARG0 contains "gain", "lose", or "create" depending on what actually happened
  - ◆ ARG1 contains a reference to the session
  - ◆ When ARG0 is "create", ARG2 contains the return value from the session's `_start` state
- `_parent`

# Session Pre-defined Events

- `_start/_stop`
  - ◆ Called when the session starts and stops
- `_child`
  - ◆ Called when a child session is created or dies
  - ◆ ARG0 contains "gain", "lose", or "create" depending on what actually happened
  - ◆ ARG1 contains a reference to the session
  - ◆ When ARG0 is "create", ARG2 contains the return value from the session's `_start` state
- `_parent`
  - ◆ When a session receives `_child`, all its current children (except the one causing `_child`) receive an `_parent` event

# Session Pre-defined Events

- `_start/_stop`
  - ◆ Called when the session starts and stops
- `_child`
  - ◆ Called when a child session is created or dies
  - ◆ ARG0 contains "gain", "lose", or "create" depending on what actually happened
  - ◆ ARG1 contains a reference to the session
  - ◆ When ARG0 is "create", ARG2 contains the return value from the session's `_start` state
- `_parent`
  - ◆ When a session receives `_child`, all its current children (except the one causing `_child`) receive an `_parent` event
- `_default`

# Session Pre-defined Events

- `_start/_stop`
  - ◆ Called when the session starts and stops
- `_child`
  - ◆ Called when a child session is created or dies
  - ◆ ARG0 contains "gain", "lose", or "create" depending on what actually happened
  - ◆ ARG1 contains a reference to the session
  - ◆ When ARG0 is "create", ARG2 contains the return value from the session's `_start` state
- `_parent`
  - ◆ When a session receives `_child`, all its current children (except the one causing `_child`) receive an `_parent` event
- `_default`
  - ◆ As we have seen, use `_default` for debugging, or sometimes catching multiple events



# Broadcasting Events to Sessions





# Broadcasting Events to Sessions

- ◆ Example: IRC bot factory - needs to tell all bots to reload their config

# Broadcasting Events to Sessions

- ◆ Example: IRC bot factory - needs to tell all bots to reload their config

```
use POE;
$|++;

my %SESSIONS;
for (1..10) {
    POE::Session->create(
        inline_states => {
            _start => sub {
                $_[KERNEL]->alias_set("session_$_");
                $SESSIONS{"session_$_"}++;
            },
            reload_config => \&reload_config,
        });
}

POE::Session->create(
    inline_states => {
        _start => sub {
            $_[KERNEL]->post($_, "reload_config") for keys %SESSIONS;
        }
    });

$poe_kernel->run();
exit(0);

sub reload_config {
    warn($_[KERNEL]->alias_list($_[SESSION]), " reloading config.\n");
    # ...
}
```

# Broadcasting Events to Sessions

- ◆ Example: IRC bot factory - needs to tell all bots to reload their config

```
use POE;
$|++;

my %SESSIONS;
for (1..10) {
    POE::Session->create(
        inline_states => {
            _start => sub {
                $_[KERNEL]->alias_set("session_$_");
                $SESSIONS{"session_$_"}++;
            },
            reload_config => \&reload_config,
        });
}

POE::Session->create(
    inline_states => {
        _start => sub {
            $_[KERNEL]->post($_, "reload_config") for keys %SESSIONS;
        }
    });

$poe_kernel->run();
exit(0);

sub reload_config {
    warn($_[KERNEL]->alias_list($_[SESSION]), " reloading config.\n");
    # ...
}
```





# **Advanced POE - Understanding Threading Issues**



# Blocking/Looping Functions



# Blocking/Looping Functions

- Many things we normally do would block POE:

# Blocking/Looping Functions

- Many things we normally do would block POE:

```
sub long_task {
  print "Working... .";
  my $counter = 0;
  while ($counter < 1_000_000) {
    print "\b", substr( "|/-\\", $counter % 4 );
    $counter++;
  }
  print "\bdone!\n";
}
```

# Blocking/Looping Functions

- Many things we normally do would block POE:

```
sub long_task {  
  print "Working... .";  
  my $counter = 0;  
  while ($counter < 1_000_000) {  
    print "\b", substr( "|/-\\", $counter % 4 );  
    $counter++;  
  }  
  print "\bdone!\n";  
}
```

- POE's kernel doesn't get a look in until "done"

# Blocking/Looping Functions

- Many things we normally do would block POE:

```
sub long_task {  
  print "Working... .";  
  my $counter = 0;  
  while ($counter < 1_000_000) {  
    print "\b", substr( "|/-\\", $counter % 4 );  
    $counter++;  
  }  
  print "\bdone!\n";  
}
```

- POE's kernel doesn't get a look in until "done"
- Need to break into smaller parts



# Blocking/Looping Functions (cont.)



# Blocking/Looping Functions (cont.)

```
sub long_task_start {
    my ($kernel, $heap) = @_[KERNEL, HEAP];
    print "Working....";
    $heap->{counter} = 0;
    $kernel->yield("long_task_continue");
}

sub long_task_continue {
    my ($kernel, $heap) = @_[KERNEL, HEAP];
    if ($heap->{counter} < 1_000_000) {
        print "\b", substr( "|/-\\", $heap->{counter} % 4 );
        $heap->{counter}++;
        $kernel->yield("long_task_continue");
    }
    else {
        print "\bdone!\n";
    }
}
```

# Blocking/Looping Functions (cont.)

```
sub long_task_start {
    my ($kernel, $heap) = @_[KERNEL, HEAP];
    print "Working....";
    $heap->{counter} = 0;
    $kernel->yield("long_task_continue");
}

sub long_task_continue {
    my ($kernel, $heap) = @_[KERNEL, HEAP];
    if ($heap->{counter} < 1_000_000) {
        print "\b", substr( "|/-\\", $heap->{counter} % 4 );
        $heap->{counter}++;
        $kernel->yield("long_task_continue");
    }
    else {
        print "\bdone!\n";
    }
}
```

- Note that this may be slow, so worthwhile becoming less granular, e.g. do 100 loops then yield



**Now the tough bit...**

# Now the tough bit...

- What was wrong with that?

# Now the tough bit...

- What was wrong with that?

```
sub long_task_start {
  my ($kernel, $heap) = @_[KERNEL, HEAP];
  print "Working....";
  $heap->{counter} = 0;
  $kernel->yield("long_task_continue");
}

sub long_task_continue {
  my ($kernel, $heap) = @_[KERNEL, HEAP];
  if ($heap->{counter} < 1_000_000) {
    print "\b", substr( "|/-\\", $heap->{counter} % 4 );
    $heap->{counter}++;
    $kernel->yield("long_task_continue");
  }
  else {
    print "\bdone!\n";
  }
}
```

# Now the tough bit...

- What was wrong with that?

```
sub long_task_start {
    my ($kernel, $heap) = @_[KERNEL, HEAP];
    print "Working....";
    $heap->{counter} = 0;
    $kernel->yield("long_task_continue");
}

sub long_task_continue {
    my ($kernel, $heap) = @_[KERNEL, HEAP];
    if ($heap->{counter} < 1_000_000) {
        print "\b", substr( "|/-\\", $heap->{counter} % 4 );
        $heap->{counter}++;
        $kernel->yield("long_task_continue");
    }
    else {
        print "\bdone!\n";
    }
}
```

- What if something calls `$kernel->yield("long_task_start")` while it is looping?



# Safe Looping Example



# Safe Looping Example

- Need to create a session for everything that can't be entered twice

# Safe Looping Example

- Need to create a session for everything that can't be entered twice

```
sub long_task_start {
my ($kernel, $heap) = @_[KERNEL, HEAP];
print "Working....";
POE::Session->create(
    inline_states => {
        _start => sub {
            $_[HEAP]->{counter} = 0;
            $_[KERNEL]->yield("long_task_continue")
        },
        long_task_continue => \&long_task_continue,
    });
}

sub long_task_continue {
my ($kernel, $heap) = @_[KERNEL, HEAP];
if ($heap->{counter} < 1_000_000) {
    print "\b", substr( "|/-\\", $heap->{counter} % 4 );
    $heap->{counter}++;
    $kernel->yield("long_task_continue");
}
else {
    print "\bdone!\n";
}
}
```



# Porting Blocking Code



# Porting Blocking Code

- Often we may need to incorporate old code into a POE framework

# Porting Blocking Code

- Often we may need to incorporate old code into a POE framework
  - ◆ But we don't want that code to block our other POE sessions

# Porting Blocking Code

- Often we may need to incorporate old code into a POE framework
  - ◆ But we don't want that code to block our other POE sessions
- Split into smaller unique components - call next sub with `yield()`



# Porting Blocking Code

- Often we may need to incorporate old code into a POE framework
  - ◆ But we don't want that code to block our other POE sessions
- Split into smaller unique components - call next sub with yield()
- Use `POE::Wheel::Run`

# Porting Blocking Code

- Often we may need to incorporate old code into a POE framework
  - ◆ But we don't want that code to block our other POE sessions
- Split into smaller unique components - call next sub with yield()
- Use POE::Wheel::Run

```
$heap->{my_program} = POE::Wheel::Run->new(  
    Program => "somescript.pl",  
    ErrorEvent => "error",  
    CloseEvent => "finished_downloading",  
    StdoutEvent => "output",  
    StderrEvent => "err_output",  
);
```

# Porting Blocking Code

- Often we may need to incorporate old code into a POE framework
  - ◆ But we don't want that code to block our other POE sessions
- Split into smaller unique components - call next sub with yield()
- Use POE::Wheel::Run

```
$heap->{my_program} = POE::Wheel::Run->new(  
    Program => "somescript.pl",  
    ErrorEvent => "error",  
    CloseEvent => "finished_downloading",  
    StdoutEvent => "output",  
    StderrEvent => "err_output",  
);
```

- Migrate I/O parts of your code to POE, so that POE can re-schedule other events while your I/O happens



# Advanced POE - Filters

# What's a filter?



# What's a filter?

- All network examples so far used line by line input



# What's a filter?

- All network examples so far used line by line input
- This is great for some protocols, but terrible for others



# What's a filter?

- All network examples so far used line by line input
- This is great for some protocols, but terrible for others
- And some protocols need to mix methods - e.g. HTTP



# What's a filter?

- All network examples so far used line by line input
- This is great for some protocols, but terrible for others
- And some protocols need to mix methods - e.g. HTTP
- This is why we use filters



# What's a filter?

- All network examples so far used line by line input
- This is great for some protocols, but terrible for others
- And some protocols need to mix methods - e.g. HTTP
- This is why we use filters
  - ◆ The default filter being "read one line"



# What's a filter?

- All network examples so far used line by line input
- This is great for some protocols, but terrible for others
- And some protocols need to mix methods - e.g. HTTP
- This is why we use filters
  - ◆ The default filter being "read one line"
- **Filters turn raw I/O into *records***





# Filters Code



# Filters Code

```
use POE;
use POE::Component::Server::TCP;

POE::Session->create(
  inline_states => {
    _start => sub {
      $_[HEAP]->{server} = POE::Component::Server::TCP->new(
        Port          => 5555,
        Address       => '127.0.0.1',
        ClientInput   => sub { $_[HEAP]->{client}->put($_[ARG0]) },
        ClientFilter  => POE::Filter::Line->new(),
      );
    },
  },
);

$poe_kernel->run();
exit(0);
```

# Filters Code

```
use POE;
use POE::Component::Server::TCP;

POE::Session->create(
  inline_states => {
    _start => sub {
      $_[HEAP]->{server} = POE::Component::Server::TCP->new(
        Port          => 5555,
        Address       => '127.0.0.1',
        ClientInput   => sub { $_[HEAP]->{client}->put($_[ARG0]) },
        ClientFilter  => POE::Filter::Line->new(),
      );
    },
  },
);

$poe_kernel->run();
exit(0);
```

- We can change this filter at will

# Filters Code

```
use POE;  
use POE::Component::Server::TCP;  
  
POE::Session->create(  
  inline_states => {  
    _start => sub {  
      $_[HEAP]->{server} = POE::Component::Server::TCP->new(  
        Port           => 5555,  
        Address        => '127.0.0.1',  
        ClientInput    => sub { $_[HEAP]->{client}->put($_[ARG0]) },  
        ClientFilter   => POE::Filter::Line->new(),  
      );  
    },  
  },  
);  
  
$poe_kernel->run();  
exit(0);
```

- We can change this filter at will
- Even mid-session



# Filters Example - HTTP





# Filters Example - HTTP

- A http server needs two filters:

# Filters Example - HTTP

- A http server needs two filters:
  - ◆ Something to extract the headers

# Filters Example - HTTP

- A http server needs two filters:
  - ◆ Something to extract the headers
  - ◆ Something for the body (e.g. POST requests)

# Filters Example - HTTP

- A http server needs two filters:
  - ◆ Something to extract the headers
  - ◆ Something for the body (e.g. POST requests)

```
POST /wiki/view/AxKit/DefaultPage HTTP/1.1
User-Agent: Mozilla/5.0 (compatible; Konqueror/3; Linux)
Referer: http://axkit.org/wiki/view/AxKit/DefaultPage?action=edit
Pragma: no-cache
Cache-control: no-cache
Accept: text/*, image/jpeg, image/png, image/*, */*
Accept-Encoding: x-gzip, gzip, identity
Accept-Charset: iso-8859-1, utf-8;q=0.5, *;q=0.5
Accept-Language: en
Host: axkit.org
Content-Type: application/x-www-form-urlencoded
Content-Length: 881
```

```
action=save&text=%3Dhead1+AxKit+Wiki%0D%0A%0D%0AWelcome+to+the+AxKit+L%3CWiki%3
E.+This+is+a+work+in+progress.%0D%0A%0D%0AThe+core+L%3CAxKit+Docs%7CAxKitDocs%3
E+are+all+online+here.+There+is+also+a%0D%0AL%3CAxKit+configuration+example%7CA
xKitConfigurationExample%3E+here+which%0D%0Ashould+give+you+a+quick+and+useful+
default+setup.%0D%0A%0D%0AOther+things+you+might+be+interested+in+are+Barrie+Sl
aymaker%27s+articles+on+perl.com%3A%0D%0A%0D%0A%3Dover%0D%0A%0D%0A%3Ditem+Intro
ducing+AxKit%0D%0A%0D%0AL%3Chttp%3A%2F%2Fwww.perl.com%2Fpub%2Fa%2F2002%2F03%2F1
2%2Faxkit.html%3E%0D%0A%0D%0A%3Ditem+XSP%2C+Taglibs+and+Pipelines%0D%0A%0D%0AL%
3Chttp%3A%2F%2Fwww.perl.com%2Fpub%2Fa%2F2002%2F04%2F16%2Faxkit.html%3E%0D%0A%0D
%0A%3Dback%0D%0A%0D%0A%3Dhr%0D%0A%0D%0AIf+you%27re+here+to+play+with+POD+Wiki+e
diting%2C+please+play+at+L%3Cthis%7CPlayPen%3E+link.+Thanks%21%0D%0A%0D%0A%3Dcu
t&texttype=1
```



# HTTP



# HTTP

- First you get the request line. This is a single line. Always.



# HTTP

- First you get the request line. This is a single line. Always.
- HTTP Headers come a line at a time

# HTTP

- First you get the request line. This is a single line. Always.
- HTTP Headers come a line at a time
- Except when they have a continuation line
  - ◆ Continuations lines are whitespace indented lines following a header that get appended to the previous line (minus the indentation)



# HTTP

- First you get the request line. This is a single line. Always.
- HTTP Headers come a line at a time
- Except when they have a continuation line
  - ◆ Continuations lines are whitespace indented lines following a header that get appended to the previous line (minus the indentation)
- What we really want to do is read the header up to the blank line

# HTTP

- First you get the request line. This is a single line. Always.
- HTTP Headers come a line at a time
- Except when they have a continuation line
  - ◆ Continuations lines are whitespace indented lines following a header that get appended to the previous line (minus the indentation)
- What we really want to do is read the header up to the blank line
- Then read the body as a stream

# HTTP

- First you get the request line. This is a single line. Always.
- HTTP Headers come a line at a time
- Except when they have a continuation line
  - ◆ Continuations lines are whitespace indented lines following a header that get appended to the previous line (minus the indentation)
- What we really want to do is read the header up to the blank line
- Then read the body as a stream
- That's three filters, to do it right

# HTTP

- First you get the request line. This is a single line. Always.
- HTTP Headers come a line at a time
- Except when they have a continuation line
  - ◆ Continuations lines are whitespace indented lines following a header that get appended to the previous line (minus the indentation)
- What we really want to do is read the header up to the blank line
- Then read the body as a stream
- That's three filters, to do it right
- But we can get by with two



# Changing Filters mid-session



# Changing Filters mid-session

- Filters are a property of the ReadWrite Wheel

# Changing Filters mid-session

- Filters are a property of the ReadWrite Wheel
- Remember, wheels must be stored in the session heap:

# Changing Filters mid-session

- Filters are a property of the ReadWrite Wheel
- Remember, wheels must be stored in the session heap:

```
$heap->{readwrite} = POE::Wheel::ReadWrite->new(  
    Handle => $socket,  
    Driver => POE::Driver::SysRW->new(),  
    Filter => POE::Filter::Line->new(),  
    InputEvent => 'socket_input',  
    ErrorEvent => 'socket_failed',  
    FlushedEvent => 'check_shutdown',  
);
```

# Changing Filters mid-session

- Filters are a property of the ReadWrite Wheel
- Remember, wheels must be stored in the session heap:

```
$heap->{readwrite} = POE::Wheel::ReadWrite->new(  
    Handle => $socket,  
    Driver => POE::Driver::SysRW->new(),  
    Filter => POE::Filter::Line->new(),  
    InputEvent => 'socket_input',  
    ErrorEvent => 'socket_failed',  
    FlushedEvent => 'check_shutdown',  
);
```

- So in every event we have access to the wheel

# Changing Filters mid-session

- Filters are a property of the ReadWrite Wheel
- Remember, wheels must be stored in the session heap:

```
$heap->{readwrite} = POE::Wheel::ReadWrite->new(  
    Handle => $socket,  
    Driver => POE::Driver::SysRW->new(),  
    Filter => POE::Filter::Line->new(),  
    InputEvent => 'socket_input',  
    ErrorEvent => 'socket_failed',  
    FlushedEvent => 'check_shutdown',  
);
```

- So in every event we have access to the wheel
- And if we have access to the wheel, we have access to the filter

# Changing Filters mid-session

- TCP Echo Server again...

```
sub socket_input {
  my ($heap, $buf) = @_[HEAP, ARG0];
  if (!$heap->{block_mode}) {
    if ($buf =~ /^BLOCK\s+(\d+)\s*$/) {
      # read a block
      my $size = $1;
      $heap->{block_mode} = 1;
      $heap->{readwrite}->set_filter(
        POE::Filter::Block->new( BlockSize => $size )
      );
    }
    elsif ($buf eq "QUIT") {
      $heap->{shutdown_now} = 1;
    }
    else {
      $heap->{readwrite}->put($buf);
    }
  }
  else {
    print "Got block: $buf\n";
    $heap->{block_mode} = 0;
    $heap->{readwrite}->set_filter( POE::Filter::Line->new() );
  }
}
```





# Back to HTTP



# Back to HTTP

- First filter must read into a request structure



# Back to HTTP

- First filter must read into a request structure
  - ◆ Provide the request type: GET/POST etc



# Back to HTTP

- First filter must read into a request structure
  - ◆ Provide the request type: GET/POST etc
  - ◆ Provide the request URI



# Back to HTTP

- First filter must read into a request structure
  - ◆ Provide the request type: GET/POST etc
  - ◆ Provide the request URI
  - ◆ Provide the HTTP version



# Back to HTTP

- First filter must read into a request structure
  - ◆ Provide the request type: GET/POST etc
  - ◆ Provide the request URI
  - ◆ Provide the HTTP version
  - ◆ And all the headers

# Back to HTTP

- First filter must read into a request structure
  - ◆ Provide the request type: GET/POST etc
  - ◆ Provide the request URI
  - ◆ Provide the HTTP version
  - ◆ And all the headers
- Second filter can be the standard POE stream filter



# Anatomy of a Filter



# Anatomy of a Filter

- Filters are easy



# Anatomy of a Filter

- Filters are easy
- A class, with three methods: new, get and put



# Anatomy of a Filter

- Filters are easy
- A class, with three methods: new, get and put
- We can ignore put() since we only want an input filter (for now)



# Anatomy of a Filter

- Filters are easy
- A class, with three methods: new, get and put
- We can ignore put() since we only want an input filter (for now)
- get() turns a byte stream into an array ref of records



# Anatomy of a Filter

- Filters are easy
- A class, with three methods: new, get and put
- We can ignore put() since we only want an input filter (for now)
- get() turns a byte stream into an array ref of records
- We want one record. So we simply need to buffer until `/^$/m`

# HTTP Filter

```
package POE::Filter::HTTPIncoming;

sub new {
    my $class = shift;
    bless { http_rec = MyHTTPRec->new(),
           buffer => '',
           finished => 0 }, $class;
}

sub get {
    my ($self, $stream) = @_;
    die "You should have switched filters" if $self->{finished};
    $stream =~ /\r/g;
    if ($stream =~ /^(*?)\n\n(.*?)$/m) {
        $self->{buffer} .= $1;
        $self->{left_over} = $2;
        $self->{finished} = 1;
        return [ $self->parse_headers() ];
    }
    $self->{buffer} .= $stream;
}

sub put {
    return $_[1];
}

sub get_pending {
    my $self = shift;
    return $self->{left_over};
}
```

# HTTP Filter (cont.)

```
sub parse_headers {
    my $self = shift;
    my $request = $self->{http_rec};
    my $buffer = $self->{buffer};

    #           POST      /foo/bar      HTTP/1.1
    if ($buffer !~ /\A([A-Z]+\s+([\n]*)\s+(HTTP\/(1\.\d)))(.*)\z/m) {
        $request->code(505); # HTTP Version Not Supported
        return $request;
    }

    my ($type, $uri, $ver, $heads) = ($1, $2, $4, $5);
    $request->type($type);
    $request->uri($uri);
    $request->http_version($ver);

    $heads =~ /\n\s+//g; # remove continuations
    foreach my $head (split(/\n/, $heads)) {
        my ($key, $val) = split(/: /, $head, 2);
        $request->add_header($key, $val);
    }

    return $request;
}
```

# Putting our filter in action

```
use POE;
use POE::Component::Server::TCP;

POE::Session->create(
    inline_states => {
        _start => sub {
            $_[HEAP]->{server} = POE::Component::Server::TCP->new(
                Port      => 8080,
                Address   => '127.0.0.1',
                ClientInput => \&input,
                ClientFilter => POE::Filter::HTTPIncoming->new(),
            );
        }});

$poe_kernel->run();

sub input {
    my ($kernel, $heap, $rec) = @_[KERNEL, HEAP, ARG0];
    warn("Got input: $rec\n");
    if (ref($rec) eq 'MyHTTPRec') {
        if ($rec->header('Content-Length')) {
            $heap->{client}->set_input_filter(
                POE::Filter::Block->new(
                    BlockSize => $rec->header('Content-Length')
                )
            );
            return;
        }
        else {
            $heap->{client}->put($response);
        }
    }
    else {
        print ("Got POSTed: $rec\n");
        $heap->{client}->put($response);
    }
    $kernel->yield('shutdown');
}
```





# POE::Filter::HTTPD



# POE::Filter::HTTPD

- Oops, we just reinvented a wheel



# POE::Filter::HTTPD

- Oops, we just reinvented a wheel
- POE::Filter::HTTPD does this for us, and does it more correctly



# POE::Filter::HTTPD

- Oops, we just reinvented a wheel
- POE::Filter::HTTPD does this for us, and does it more correctly
- Ships with POE



# Filters Roundup





# Filters Roundup

- Filters are just *objects* that support a *get()* and a *put* method

# Filters Roundup

- Filters are just *objects* that support a *get()* and a *put* method
- Filters may also support a *get\_pending()* method to get pending buffered data when switching filters

# Filters Roundup

- Filters are just *objects* that support a *get()* and a *put* method
- Filters may also support a *get\_pending()* method to get pending buffered data when switching filters
- **Filters just turn a stream of bytes into records**



# **Advanced POE - Alternate Event Loops**



# Why use Alternate Event Loops?





# Why use Alternate Event Loops?

- Default event loop is `select()` based



# Why use Alternate Event Loops?

- Default event loop is `select()` based
- `select()` can listen only to a certain number of filehandles (`FD_SETSIZE`)

# Why use Alternate Event Loops?

- Default event loop is `select()` based
- `select()` can listen only to a certain number of filehandles (`FD_SETSIZE`)
  - ◆ On my Linux box this is 1024



# Why use Alternate Event Loops?

- Default event loop is `select()` based
- `select()` can listen only to a certain number of filehandles (`FD_SETSIZE`)
  - ◆ On my Linux box this is 1024
- You might need to integrate some current code using `Event.pm`

# Why use Alternate Event Loops?

- Default event loop is `select()` based
- `select()` can listen only to a certain number of filehandles (`FD_SETSIZE`)
  - ◆ On my Linux box this is 1024
- You might need to integrate some current code using `Event.pm`
- You are writing a GUI program

# Why use Alternate Event Loops?

- Default event loop is `select()` based
- `select()` can listen only to a certain number of filehandles (`FD_SETSIZE`)
  - ◆ On my Linux box this is 1024
- You might need to integrate some current code using `Event.pm`
- You are writing a GUI program
- Enable an alternate event loop by loading it prior to POE:

# Why use Alternate Event Loops?

- Default event loop is select() based
- select() can listen only to a certain number of filehandles (FD\_SETSIZE)
  - ◆ On my Linux box this is 1024
- You might need to integrate some current code using Event.pm
- You are writing a GUI program
- Enable an alternate event loop by loading it prior to POE:

```
use IO::Poll;  
use POE;
```

# IO::Poll



# IO::Poll

- The poll() syscall is an alternative to select()



# IO::Poll

- The poll() syscall is an alternative to select()
- It scales better than select in certain cases, on certain OSes, when the wind is blowing right.



# IO::Poll

- The poll() syscall is an alternative to select()
- It scales better than select in certain cases, on certain OSes, when the wind is blowing right.
- Then again, it may be slower. Test it.



# Event.pm



# Event.pm

- Use when porting code that uses Event.pm already



# Event.pm

- Use when porting code that uses Event.pm already
- Also use when you need safe signals on perls < 5.8.0





**Tk**

# Tk

```
use Tk;
use POE;

POE::Session->create(
    inline_states => {
        _start    => \&ui_start,
        ev_count  => \&ui_count,
        ev_clear  => \&ui_clear,
    }
);

$poe_kernel->run();

sub ui_start {
    my ($kernel, $session, $heap) = @_[KERNEL, SESSION, HEAP];

    $poe_main_window->Label( -text => 'Counter' )->pack;

    $heap->{counter_widget} =
        $poe_main_window->Label( -textvariable => \$$heap->{counter} )->pack;

    $poe_main_window->Button(
        -text => 'Clear',
        -command => $session->postback( 'ev_clear' )
    )->pack;

    $kernel->yield("ev_count");
}

sub ui_count {
    $_[HEAP]->{counter}++;
    $_[KERNEL]->yield("ev_count");
}

sub ui_clear {
    $_[HEAP]->{counter} = 0;
}
```



# Components

**<curl> components are cool.**



# POE Components



# POE Components

- Components generally encapsulate functionality in a session

# POE Components

- Components generally encapsulate functionality in a session
- Example: `POE::Component::IRC` - encapsulates an IRC client in a session

# POE Components

- Components generally encapsulate functionality in a session
- Example: POE::Component::IRC - encapsulates an IRC client in a session
- All a Component really does is wrap most of the session setup and wheel processing "mess"



# POE Components

- Components generally encapsulate functionality in a session
- Example: `POE::Component::IRC` - encapsulates an IRC client in a session
- All a Component really does is wrap most of the session setup and wheel processing "mess"
- Components range from protocol specific components, to enhancements to POE's core capabilities

# POE Components

- Components generally encapsulate functionality in a session
- Example: POE::Component::IRC - encapsulates an IRC client in a session
- All a Component really does is wrap most of the session setup and wheel processing "mess"
- Components range from protocol specific components, to enhancements to POE's core capabilities
- Often referred to using the short name "PoCo" - e.g. PoCoIRC

# POE Components

- Components generally encapsulate functionality in a session
- Example: POE::Component::IRC - encapsulates an IRC client in a session
- All a Component really does is wrap most of the session setup and wheel processing "mess"
- Components range from protocol specific components, to enhancements to POE's core capabilities
- Often referred to using the short name "PoCo" - e.g. PoCoIRC
- Components we'll discuss here:
  - ◆ POE::Component::IRC
  - ◆ POE::Component::Server::TCP
  - ◆ POE::Component::Client::TCP
  - ◆ POE::Component::DBIAgent
  - ◆ POE::Component::IKC
  - ◆ POE::Component::Client::DNS

# POE::Component::IRC

```
use POE;
use POE::Component::IRC;
use Chatbot::Eliza;

# Initialize Chatbot::Eliza
my $chatbot = Chatbot::Eliza->new();

# One PoCoIRC instance and one session per bot
# The parameter is a session alias
POE::Component::IRC->new("doctor");

POE::Session->create(
    inline_states => {
        _start => \&bot_start,
        irc_001   => \&on_connect,
        irc_public => \&on_public,
        irc_join  => \&on_join,
    }
);

$poe_kernel->run();
exit(0);

sub bot_start {
    my ($kernel, $heap, $session) = @_[KERNEL, HEAP, SESSION];

    $kernel->post(doctor => register => "all");

    $kernel->post(doctor =>
        connect => {
            Nick => "doctor",
            Username => "doctorbot",
            Ircname => "POE::Component::IRC + Eliza demo",
            Server => "grou.ch",
            Port => 6667,
        });
}
```

# POE::Component::IRC (cont.)

```
sub on_connect {
    $_[KERNEL]->post( doctor => join => "#elizabot" );
    print "Joined channel #elizabot\n";
}

sub on_public {
    my ($kernel, $who, $where, $msg) =
        @_[KERNEL, ARG0, ARG1, ARG2];

    $msg =~ s/^doctor[:,]?\s+//;

    my ($nick, undef) = split(/!/, $who, 2);
    my $channel = $where->[0];

    my $response = $chatbot->transform($msg);
    $kernel->post(doctor => privmsg => $channel, "$nick: $response");
}

sub on_join {
    my ($kernel, $who, $channel) = @_[KERNEL, ARG0, ARG1];

    my ($nick, undef) = split(/!/, $who, 2);
    $kernel->post(doctor => privmsg => $channel, "$nick: How can I help you?");
}
```

# POE::Component::IRC Events

- `irc_connected` - Called on successful connection
- `irc_disconnected` - Called when you are disconnected
- `irc_join` - Called when someone joins a channel - useful for op-bots
- `irc_public` - For when normal text is sent to a channel
- `irc_msg` - Sent when text is sent to the bot privately via `/msg`
- `irc_part` - Called when a user leaves the channel - useful for limit-bots

# POE::Component::IRC::Object

```
package ElizaBot;
use Chatbot::Eliza;
use POE;
use POE::Component::IRC::Object;
use base qw(POE::Component::IRC::Object);
my $chatbot = Chatbot::Eliza->new();

sub irc_001 {
    $_[OBJECT]->join( "#elizabot" );
}

sub irc_public {
    my ($self, $kernel, $who, $where, $msg) =
        @_[OBJECT, ARG0, ARG1, ARG2];

    $msg =~ s/^doctor[:,]?\s+//;
    my ($nick, undef) = split(/!/, $who, 2);
    my $channel = $where->[0];

    my $response = $chatbot->transform($msg);
    $self->privmsg( $channel, "$nick: $response" );
}

package main;

ElizaBot->new(
    Nick => 'doctor',
    Server => 'grou.ch',
    Port => 6667,
);

$poe_kernel->run();
exit(0);
```



# POE::Component::Client::TCP



# POE::Component::DBIAgent



# POE::Component::DBIAgent

- Runs Asynchronous DBI calls

# POE::Component::DBIAgent

- Runs Asynchronous DBI calls
- Calls an event when finished



# POE::Component::DBIAgent

- Runs Asynchronous DBI calls
- Calls an event when finished
- Uses helper processes and POE::Wheel::Run



# DBIAgent Example



# DBIAgent Example

```
sub start_handler {
    # in some event handler (probably _start)
    $_[HEAP]->{helper} = POE::Component::DBIAgent->new(
        DSN => [ $dsn, $username, $password ],
        Queries => $self->make_queries,
        Debug => 1,
    );
}

sub get_dogs {
    # then execute a query any time:
    $_[HEAP]->{helper}->query(get_dogs => $_[SESSION] => dogs_ca
}

sub dogs_callback {
    my ($heap, $data) = @_[HEAP, ARG0];

    # $data is an array ref of array refs.
}
```



# POE::Component::Client::DNS



# POE::Component::Client::DNS

- Does async DNS queries

# POE::Component::Client::DNS

- Does async DNS queries

```
POE::Component::Client::DNS->spawn(
    Alias      => 'named',
    Timeout    => 120,
);

$kernel->post(
    named => resolve => dns_handler =>
    $address,      # the address to resolve
    'A', 'IN'     # the record type and class to return
);

sub dns_handler {
    my (@original_request_parameters) = @{$_[ARG0]};
    my ($net_dns_packet, $net_dns_errorstring) = @{$_[ARG1]};

    my $request_address = $original_request_parameters[0];

    return unless (defined $net_dns_packet);

    my @net_dns_answers = $net_dns_packet->answer;
    return unless (@net_dns_answers);

    foreach my $net_dns_answer (@net_dns_answers) {
        printf( "%25s (%-10.10s) %s\n",
                $request_address,
                $net_dns_answer->type,
                $net_dns_answer->rdatastr
            );
    }
}
```



# Thank You

- Further Resources
  - ◆ <http://poe.perl.org> - home of POE
  - ◆ <http://search.cpan.org/> - to find POE modules
  - ◆ IRC: #poe on irc.rhizomatic.net
- Special Thanks to Rocco for answering all my dumb questions